

Cours 4

programmation dynamique

La programmation dynamique c'est une méthode de calcul d'une solution optimale à un problème d'optimisation vérifiant deux propriétés

- « **décomposition récursive** » : le problème initial se décompose en sous-problèmes (avec recouvrement possible)
- « **optimalité des sous-problèmes** » :
une solution optimale est construite en combinant des solutions optimales de sous-problèmes
[Principe d'optimalité de Bellman]

Intérêt : «calcul systématique» d'une **solution optimale** à partir de la caractérisation récursive de la valeur optimale.

Remarque: si les sous-problèmes sont sans recouvrement, on parle de « Diviser pour Régner »

Chap 2. Calcul d'une solution optimale par prog. dynamique

Plan

1. Exemple du sac à dos binaire

1. Rappel version avec mémoïsation
2. Remontée d'une solution optimale

2. Prog Dyn: la Méthode générale

3. Lien avec les plus courts chemins

4. Conclusion

5. Ex (hors cours) : cageots de fraises

6. TD: rendu monnaie optimal + jeu télévisé

1. Exemple : sac à dos binaire

Résolution par programmation dynamique

- ◆ Soit un sac de volume C et N articles différents
 - Données: pour $n=1..N$
 - ◆ u_n l'utilité (ou bénéfice) du $n^{\text{ième}}$ article,
 - ◆ v_n son volume
- ◆ Trouver un remplissage du sac d'utilité maximale

ie trouver (x_1, \dots, x_N) dans $\{0, 1\}^N$ qui maximise $\sum_{n=1..N} x_n \cdot u_n$ avec la contrainte $\sum_{n=1..N} x_n \cdot v_n \leq C$

Etape 1: caractérisation récursive

◆ Récursion, donc indices à « récurrer »...

- n = nombre d'objets restants à examiner ($1 \leq n \leq N$)
- V = volume libre dans le sac ($0 \leq V \leq C$)
- Déf: $\hat{U}_{V,n}$ = utilité maximale d'un sac de volume V qu'on peut remplir avec les objets 1 à n

◆ Formule récursive:

$$\hat{U}_{V,n} = \begin{cases} \max(u_n + \hat{U}_{V-v_n, n-1}, \hat{U}_{V, n-1}) & \text{si } v_n \leq V \\ \hat{U}_{V, n-1} & \text{sinon} \end{cases}$$

$$\hat{U}_{V,0} = 0$$

Prog. récursif naïf

$$\hat{U}_{V,n} = \begin{cases} \max(u_n + \hat{U}_{V-v_n, n-1}, \hat{U}_{V, n-1}) & \text{si } v_n \leq V \\ \hat{U}_{V, n-1} & \text{sinon} \end{cases}$$
$$\hat{U}_{V,0} = 0$$

```
double sacMax( int V, int n ) {  
    | double res ;  
    | if (n==0) { res = 0; }  
    | else if (v[n] ≤ V) {  
    |     max1 = u[n] + sacMax( V – v[n], n-1 ) ;  
    |     max2 = sacMax( V, n-1 ) ;  
    |     if (max1 > max2) { res = max1 ; }  
    |     else             { res = max2 ; }  
    | }  
    | else { res= sacMax( V, n-1 ) ; }  
    |  
    | return res;  
}
```

$T(n) \geq 2 T(n-1)$: inefficace car bcp de calculs redondants $\Omega(2^n)$

2. Mémoïsation

$$\hat{U}_{V,n} = \begin{cases} \max(u_n + \hat{U}_{V-v_n, n-1}, \hat{U}_{V, n-1}) & \text{si } v_n \leq V \\ \hat{U}_{V, n-1} & \text{sinon} \end{cases}$$

$$\hat{U}_{V,0} = 0$$

```
double sacMaxMemoisation( int N, int C, int* u, int* v) {  
    double memVal[...,...] ; // déclaration tableau temporaire
```

```
    double sacMax( int V, int n) { double res;  
        if (memVal[V, n] != -1) return memVal[V, n] ;  
        if (n==0) { res = 0; }  
        else if (v[n] ≤ V {  
            max1 = u[n] + sacMax( V – v[n], n-1) ;  
            max2 = sacMax( V, n-1 ) ;  
            if (max1 > max2) { res = max1 ; }  
            else { res = max2 ; }  
        }  
        else { res= sacMax( V, n-1 ) ; }  
        memVal[V, n] = res ;  
        return res;  
    }
```

// main sacMaxMemoisation

```
memVal = allocate_and_init(N , C , -1) ;  
return sacMax( C, N) ;  
}
```

2. Mémoïsation avec choix optimaux

$\hat{U}_{V,n} = \begin{cases} \max(u_n + \hat{U}_{V-v_n, n-1}, \hat{U}_{V, n-1}) & \text{si } v_n \leq V \\ 0 & \text{sinon} \end{cases}$

```
double sacMaxMemoisation( int N, int C, int* u, int* v) {  
    double memVal[...,...] ; // déclaration tableau temporaire  
    bool choixOpt[...,...] ; // choixOpt[V, n]=1 ssi mettre objet n dans V est optimal!
```

```
double sacMax( int V, int n) {  double res; bool argmax;  
| if (memVal[V, n] ≠ -1) return memVal[V, n] ;  
| if (n==0) { res = 0; }  
| else if (v[n] ≤ V {  
|     max1 = u[n] + sacMax( V – v[n], n-1 ) ;  
|     max2 = sacMax( V, n-1 ) ;  
|     if (max1 > max2) { res = max1 ; argmax= 1; }  
|     else  
|         { res = max2 ; argmax= 0; }  
|     }  
|     else { res= sacMax( V, n-1 ) ; argmax= 0; }  
|     memVal[V, n] = res ;    choixOpt[V, n] = argmax;  
|     return res;  
| }
```

// main sacMaxMemoisation

```
memVal = allocate_and_init(N , C , -1) ; choixOpt = allocate (V, N ) ;  
return sacMax( C, N ) ;  
}
```

3. Remontée solution à partir des choix optimaux précalculés

- ◆ Faut-il prendre ou non x_n si il reste un volume V ?
-> Le choix optimal pour tout (n, V) a été calculé et est mémorisé dans **choixOpt[V, n]**

- ◆ => calcul d'une solution optimale *top-down*:

```
solutionOptimale [ 1..N ] = {0, ..., 0} ;
for (V = C, n= N ; n≠0 ; --n) {
    | // on prend l'objet n dans la solution optimale en cours ssi choixOpt[V, n] ==1
    | | solutionOptimale [n] = choixOpt[ V, n ] ;
    | | if (solutionOptimale [n] == 1) V = V - v[n] ;
}
```

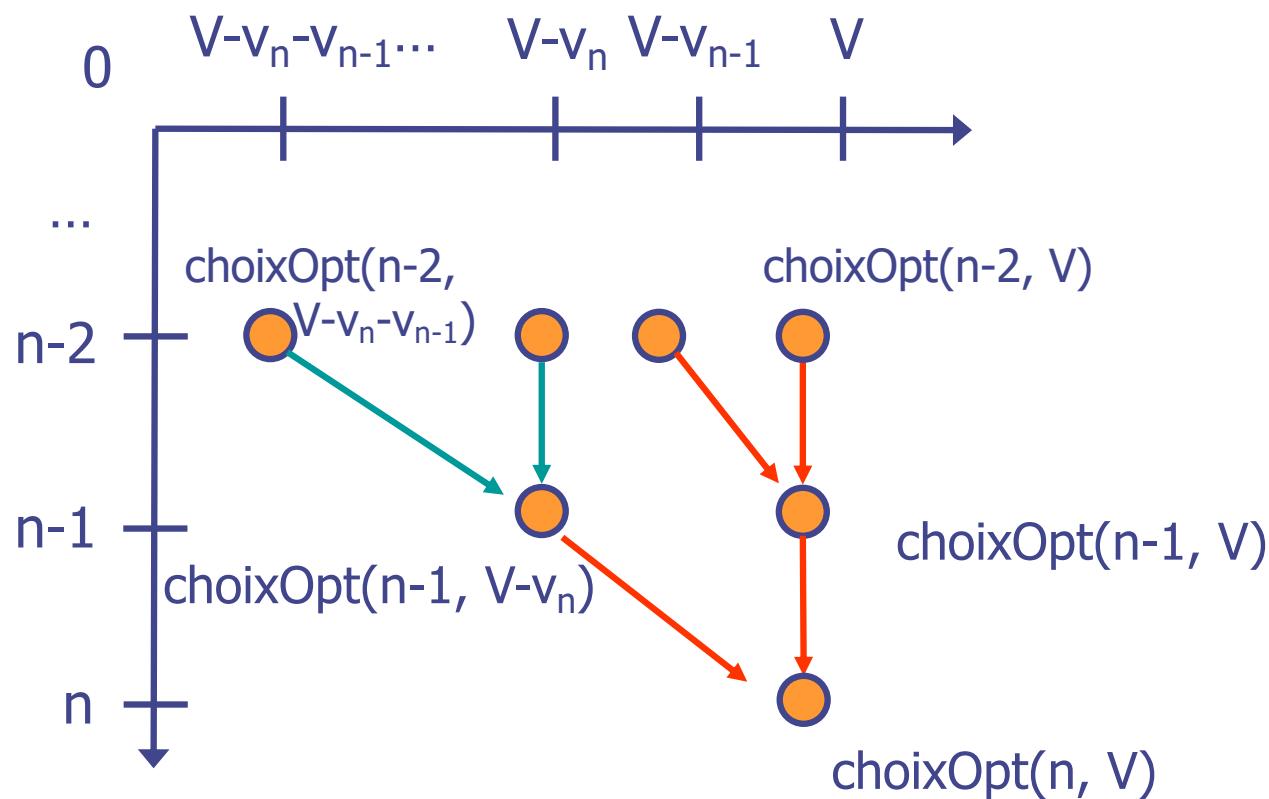
Analyse de coût du prog. récursif

- ◆ **Conclusion:** *méthode systématique de calcul d'une solution optimale à partir d'une formule récursive !*
- ◆ **Coût calcul:**
 - Précalcul des Choix optimaux : $O(N.C)$
 - Remontée solution: $O(N)$ *négligeable*
 - NB *pas plus cher que calculer la valeur optimale!*
- ◆ **Coût Mémoire:**
 1. **Espace mémoire utilisé :**
 - Stockage choix optimaux: $N.C$
 - NB: Mémoïsation peut être faite dans le tableau choix optimaux
 2. **Défauts de cache :** *Comment stocker et parcourir les tableaux?*
 - $O(N.C)$ accès à choixOpt et memVal car calcul récursif dans un ordre non contigu en mémoire => inefficace *a priori*
 - => **analyse dépendances** (objectif $O(NC / L)$ défauts de cache)

4. Analyse dépendances

- ◆ Objectif : Programme « itératif » dans l'ordre des dépendances et améliorant la mémoire/localité

$$\hat{U}_{V,n} = \begin{cases} \max(u_n + \hat{U}_{V-v_n, n-1}, \hat{U}_{V, n-1}) & \text{si } v_n \leq V \\ \hat{U}_{V, n-1} & \text{sinon} \end{cases}$$
$$\hat{U}_{V,0} = 0$$



Nombreux
ordonnancements
itératifs possibles:

- par lignes
- ou par colonnes
- ou par blocs
-

Programme itératif selon les dépendances

Exemple : par lignes

```
for (n = 1; n <= Nt; ++n) { // par lignes
    { for (V = 0 ; v <= C ; ++V) { // par colonnes
        | { if (v[n] ≤ V )
        | | { max1 = u[n] + memVal [V – v[n], n-1 ] ;
        | | | max2 = memVal [V, n-1 ];
        | | | if (max1 > max2) { res = max1 ; argmax = 1; }
        | | | else             { res = max2 ; argmax = 0; }
        | | }
        | | else { res= memVal [V, n-1 ] ; argmax = 0; }
        | | memVal [ V, n] = res ; choixOpt[V, n] = argmax ;
    } }
```

Question: comment stocker choixOpt et memVal?
choixOpt[1..N][0..C] ou **choixOpt[0..C][1..N]** ?

Programme itératif selon les dépendances

Exemple : par lignes

```
for (n = 1; n <= Nt; ++n) { // par lignes
    { for (V = 0 ; v <= C ; ++V) { // par colonnes
        | { if (v[n] ≤ V )
        | | { max1 = u[n] + memVal [n-1][ V - v[n] ] ;
        | | | max2 = memVal[ n-1 ][ V ];
        | | | if (max1 > max2) { res = max1 ; argmax = 1; }
        | | | else             { res = max2 ; argmax = 0; }
        | | }
        | | else { res= memVal [n-1] [ V ] ; argmax = 0; }
        | | memVal [ n ][ V ] = res ; choixOpt[n][ V ] = argmax ;
    } }
```

ATTENTION : ordre stockage mémoire

- Localité cache : Parcours contigü par ligne => **choixOpt[1..N][0..C]**
Nombre de défauts de cache = $O(N.C / L)$ => cache oblivious
- ! Si par colonnes: for V { for n ... } => choixOpt[0..C] [1..N]

La programmation dynamique: Méthode générale en 4 étapes

1. Caractériser récursivement la valeur optimale

- **Equation de Bellman** : le plus difficile... *Trouver les indices à récurer !*

2. Programme 1 mémorisant les valeurs optimales des sous-problèmes

- 1) **Programmation récursive avec mémoïsation** de l'équation de Bellman
avec une seule variable « res » au début et un seul « return res » en fin
- 2) **Mémoriser les choix optimaux** (dans un tableau choixOpt = ArgMax)
*NB Appels récursifs top-down, mais **ordre des calculs bottom-up** !*

3. Programme 2 qui construit une solution optimale

- En parcourant le tableau choixOpt précalculé => **ordre top-down**

4. Analyse des dépendances et programmation itérative efficace

- programme **itératif** dans l'ordre des dépendances: *bottom-up, gain mémoire*
- Gain en localité: programmation par blocs (*blocking cache-aware/oblivious*) 14

Variante : un autre exemple

- ◆ Sac à dos n-aire

1. Exemple : sac à dos n-aire

Soit un sac de volume C et N types d'objets :

- Données: pour $n=1..N$
 - ◆ u_n l'utilité (ou bénéfice) des objets de type n ,
 - ◆ v_n le volume des objets de type n

Trouver un remplissage du sac d'utilité maximale

ie trouver (x_1, \dots, x_N) dans \mathbb{N}^N qui maximise $\sum_{n=1..N} x_n \cdot u_n$ avec la contrainte $\sum_{n=1..N} x_n \cdot v_n \leq C$

Remarque: $0 \leq x_n \leq C / v_n$

1. Equation de Bellman : différentes possibilités

- Récurrence sur n (V =volume libre) :
 $F(n, V) = \text{Max} \{x_n \cdot u_n + F(n-1, V - x_n \cdot v_n) \text{ pour } x_n=0 \dots (V / v_n)\}$
Condition d'arrêt: $F(0, V) = 0$ pour tout $0 \leq V \leq C$
- Ou récurrence sur le volume libre V :
 $F(V) = \text{Max} \{u_n + F(V - v_n) \text{ pour tout } n \text{ tel que } v_n \leq V\}$
Condition d'arrêt: $F(0) = 0$
- Ou récurrence sur n et V (V =volume libre) :
 $F(n, V) = \text{Max} \{ F(n-1, V) ; u_n + F(n, V - v_n) \text{ si } v_n \leq V\}$
Condition d'arrêt: $F(0, V) = 0$ pour tout $0 \leq V \leq C$

etc

2. Programme récursif avec mémoïsation+ choix optimaux

```
double sacMaxMemoisation( int N, int C) {  
    double memVal[....,...] ; // déclaration tableau temporaire  
    bool choixOpt[....,...] ; // choixOpt[n,V]= xn, ssi mettre xn objets n dans V est optimal!  
    double sacMax( int V, int n) { double valmax ; bool argmax ;  
        | if (memVal[n, V] ≠ -1) return memVal[n, V] ;  
        | if (n==0) { valmax = 0; }  
        | else { // F(n,V) = Max {xn.un + F( n-1, V- xn.vn ) pour xn=0 .. (V / vn)}  
        |     int x = V / v[n] ;  
        |     valmax = x*u[n] + sacMax( V – x*v[n], n-1) ; argmax = x;  
        |     for ( x=x-1; x≥0; --x) {  
        |         aux = x*u[n] + sacMax( V – x*v[n], n-1) ;  
        |         if (aux > valmax) { valmax = aux; argmax = x; }  
        |     }  
        |     memVal[n, V] = valmax ; choixOpt[n,V] = argmax;  
        |     return valmax;  
    }  
    // main sacMaxMemoisation  
    memVal = allocate_and_init(N, C, -1) ; choixOpt = allocate (N, C) ;  
    return sacMax (V , N) ;  
}
```

3. Remontée solution à partir des choix optimaux précalculés

Combien met-on d'objets de type n ??

-> Le choix optimal dépend du volume V libre ;
il a été calculé et mémorisé dans **choixOpt[n, V]**

=> calcul d'une solution optimale *top-down*:

```
solutionOptimale [ 1..N] = {0, ..., 0 } ;
```

```
for (V = C, n= N; n>0 ; --n) { // Initialement, le volume libre est C
```

```
    | int x = choixOpt[n, V] ; // On met x objets de type n ans la sol. opt. en cours
```

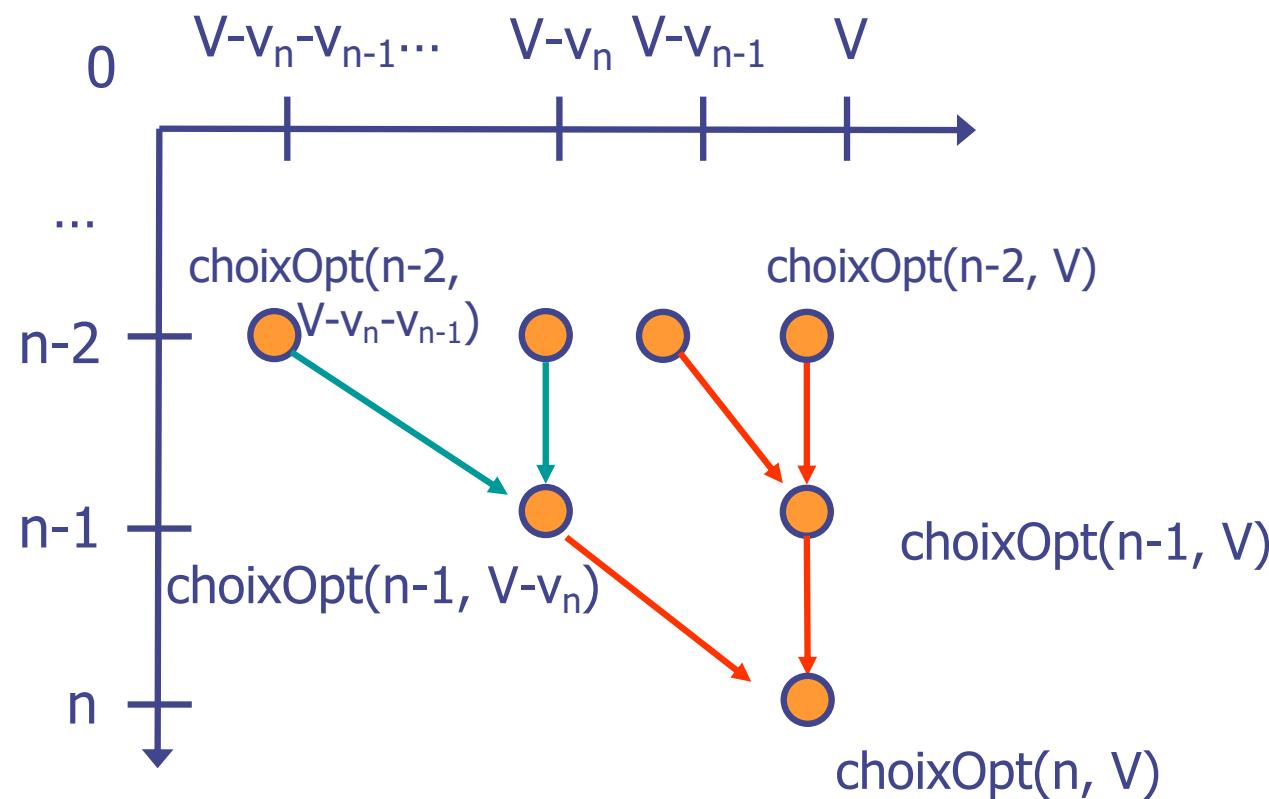
```
    | solutionOptimale [n] = x ;
```

```
    | V = V - x*v[n] ;
```

```
}
```

4. Analyse dépendances => programme itératif

- ◆ Objectif : Programme « itératif » dans l'ordre des dépendances et améliorant la mémoire/localité



Ordonnancements
Itératifs possibles:

- par lignes
- ou par colonnes
- ou par blocs
-

Programme itératif selon les dépendances

Exemple : par lignes => stockage [n] [V] dans l'ordre!

```
for (V = 0; V <= C; ++C) memVal[0][V] = 0 ; //initialisation pour n=0
```

```
for (n = 1; n <= Nt; ++n) { // par lignes
```

```
{ for (V = 0 ; v <= Ct ; ++V) { // par colonnes
```

```
| // F(n,V) = Max {x_n.u_n + F( n-1, V- x_n.v_n ) pour x_n=0 .. (V / v_n)}
```

```
| int valmax = memVal[n-1][ V] ; argmax = 0 ;
```

```
| int xcour = 1 ; int ucour =u[n]; int vcour = V - v[n] ;
```

```
| for (; (vcour >0) ; xcour+=1, ucour+=u[n], vcour -= v[n] )
```

```
| { int aux = ucour + memVal[ n-1][vcour] ;
```

```
| if (aux > valmax) { valmax = aux; argmax = xcour; }
```

```
| }
```

```
| memVal [ n][ V] = res ; choixOpt[n][ V] = choix ;
```

```
}
```

ATTENTION : ordre stockage mémoire

- Localité cache : Parcours contigü par ligne => **choixOpt[1..N][0..C]**
Nombre de défauts de cache = $O(NC / L)$ => cache oblivious
-  Si par colonnes: for v { for n ... } => **choixOpt[0..C] [1..N]**

Conclusion et analyse de coût

- ◆ **Conclusion:** *calcul systématique d'une solution optimale à partir de la formule récursive !*
- ◆ **Coût calcul:** $\sum_{V=0..C} \sum_{k=1..N} V / v[k] = O(N.C^2)$
 - On peut réduire le coût à $O(N.C)$ avec une équation de Bellman faisant moins de calcul de Max :
 $F(n,V) = \text{Max } \{ F(n-1,V) ; u_n + F(n, V - v_n) \text{ si } v_n \leq V \}$
[exercice: écrire le programme itératif avec gestion du cache]
- ◆ **Coût mémoire (avec cette nouvelle équation):**
 - Stockage choix optimaux: $O(NC)$
 - NB: Mémoïsation peut être faite dans une tableau de taille C
- ◆ **Localité :** $O(N.C / L)$ accès : optimal (à un facteur au plus 2)

Chap 2. Calcul d'une solution optimale par prog. dynamique

Plan

1. Exemple du sac à dos binaire

1. Rappel version avec mémoïsation
2. Remontée d'une solution optimale

2. Prog Dyn: la Méthode générale

3. Lien avec les plus courts chemins

4. Conclusion

5. Ex (hors cours) : cageots de fraises

6. TD: rendu monnaie optimal + jeu télévisé

La programmation dynamique: Méthode générale en 4 étapes

1. Caractériser récursivement la valeur optimale

- Equation de Bellman : le plus difficile... *Trouver les indices à récurer !*

2. Programme 1 mémorisant les valeurs optimales des sous-problèmes

- 1) Programmation récursive avec mémoïsation de l'équation de Bellman
avec une seule variable « res » au début et un seul « return res » en fin
- 2) Mémoriser les choix optimaux (dans un tableau choixOpt = ArgMax)
NB *Appels récursifs top-down, mais ordre des calculs bottom-up !*

3. Programme 2 qui construit une solution optimale

- En parcourant le tableau choixOpt précalculé => *ordre top-down*

4. Analyse des dépendances et programmation itérative efficace

- programme **itératif** dans l'ordre des dépendances: *bottom-up, gain mémoire*
- Gain en localité: programmation par blocs (*blocking cache-aware/oblivious*)

Système évoluant au cours du temps (T périodes) :

- ▶ $x_{t+1} = f_t(x_t, u_t)$ pour $t = 1, \dots, T$
- ▶ x_t = état du système en période t
- ▶ u_t = décision en période t , dépend (en général) de x_t
 - ▶ dépend potentiellement de la séquence x_1, x_2, \dots, x_t
- ▶ $f_t : \mathcal{X} \times \mathcal{U} \mapsto \mathcal{X}$ = fonction de transition en période t
 - ▶ $\mathcal{X} := \{\text{états du système}\}$, $\mathcal{U} := \{\text{décisions possibles}\}$
- ▶ $c_t(x_t, u_t)$ = coût/profit en période t
 - ▶ $c_{T+1}(x_{T+1})$ coût terminal en fin d'horizon
- ▶ **Objectif** : minimiser (ou maximiser) la somme des coûts ou profits réalisés :

$$\text{Pour } x_1 \in \mathcal{X}, \quad V_1^*(x_1) = \min_{(u_1, \dots, u_T)} \left(c_{T+1}(x_{T+1}) + \sum_{t=1}^T c_t(x_t, u_t) \right)$$

Exemple : sac à dos binaire

- ▶ Sac à dos avec une *capacité* (= volume) totale b
- ▶ n articles numérotés de 1 à n
- ▶ À chacun est associé un *profit* (opposé de coût) c_i et un *volume* a_i
 - ▶ Vecteurs de profits ${}^t c$ et de volume ${}^t a$
- ▶ *Décision* binaire pour chaque article i :
 - ▶ $x_i = 1$ si on le met dans le sac à dos
 - ▶ $x_i = 0$ sinon
- ▶ **Problème** : $\max cx : ax \leq b, x \in \{0,1\}^n$

- ▶ Décomposer les décisions : on décide d'abord pour l'objet 1 puis 2, etc.
 - ▶ Décider en période $i = 1, \dots, n$ si prendre ou non l'objet i
- ▶ Après la décision pour l'objet 1
 - ▶ Il reste un sac à dos avec $n - 1$ objets
 - ▶ Capacité b si on ne prend pas l'objet 1 et de capacité $b - a_1 \geq 0$ sinon
- ▶ ⇒ Problème de décision *séquentiel* sur les périodes $1, \dots, n$
 - ▶ États du système = capacités restantes $\mathcal{X} = \{0, \dots, b\}$
 - ▶ Actions limitées par une capacité résiduelle positive

Système évoluant au cours du temps (T périodes) :

- ▶ $x_{t+1} = f_t(x_t, u_t)$ pour $t = 1, \dots, T$
- ▶ x_t = état du système en période $t \rightsquigarrow K_i$ capacité restante
- ▶ u_t = décision en période t , dépend (en général) de $x_t \rightsquigarrow x_i$ choix ou non de l'article i
- ▶ $f_t : \mathcal{X} \times \mathcal{U} \mapsto \mathcal{X}$ = fonction de transition en période $t \rightsquigarrow K_{i+1} = K_i - a_i x_i$
 - ▶ a_i volume de l'article i
- ▶ $c_t(x_t, u_t) = \text{coût/profit en période } t \rightsquigarrow c_i x_i$
 - ▶ $c_{T+1}(x_{T+1})$ coût terminal en fin d'horizon $\rightsquigarrow 0$

- Objectif : minimiser (ou maximiser) la somme des coûts (ou profits) réalisés :

$$V_1^*(x_1) = \min_{(u_1, \dots, u_T)} c_{T+1}(x_{T+1}) + \sum_{t=1}^T c_t(x_t, u_t), \forall x_1 \in \mathcal{X}$$

- Pour le sac à dos :

$$\begin{aligned} V_1^*(b) &= \max_{\substack{(x_1, \dots, x_n) \\ \in \{0, 1\}^n : ax \leq b}} \sum_{i=1}^n c_i x_i \end{aligned}$$

Equation de Bellman

Hypothèse « sans mémoire » : pour tout t , la solution optimale à t à partir de l'état x_t est indépendante des choix précédents ($t-1, \dots$) qui ont amené à l'état x_t

Principe d'optimalité :

- **« décomposition récursive avec recouvrement » :** le problème initial se décompose en sous-problèmes (avec recouvrement)
- **« optimalité des sous-problèmes » :** une solution optimale peut être construite en combinant des solutions optimales de sous-problèmes
[Principe d'optimalité de Bellman]

Caractérisation récursive de la valeur optimale...
et donc «calcul systématique» d'une solution optimale !

Remarque: si les sous-problèmes sont sans recouvrement, on parle de « Diviser pour Régner »

Chap 2. Calcul d'une solution optimale par prog. dynamique

Plan

1. Exemple du sac à dos binaire

1. Rappel version avec mémoïsation
2. Remontée d'une solution optimale

2. Prog Dyn: la Méthode générale

3. Lien avec les plus courts chemins

4. Conclusion

5. Ex (hors cours) : cageots de fraises

6. TD: rendu monnaie optimal + jeu télévisé

Plus long chemin dans un DAG

DAG $G=(V, E)$ pondéré, complet

- **Chemin critique** = $\text{Max}_{u,v \text{ connectés}} L(u,v)$
avec $L(u,v) = \text{plus long chemin élémentaire de } u \text{ à } v$
 $L(u,v) = \text{Max}_{w \text{ succ de } u, w \text{ connecté à } v} \text{poids}(u,w)+L(w,v))$
 $L(u,u) = 0$

NB Autres problèmes similaires:

- Plus court chemin dans un DAG ? (idem)
- Plus court chemin élémentaire dans G orienté quelconque?
(Bellman-Ford)
- Plus long chemin élémentaire dans G orienté quelconque ?
(NP-dur, réduction à graphe hamiltonien)

Programmation dynamique et plus court chemin

- Un problème de programmation dynamique se ramène à un plus court chemin.
 - Exemple : Sac à dos: sommets = (n, V)
 - ◆ Les successeurs de (n, V) sont $(n-1, V)$ et $(n-1, V-v_n)$
 - ◆ Plus court chemin de $(n_{\text{init}}, V_{\text{init}})$ à $(0, \dots)$
 - La programmation dynamique « dérive » directement du problème initial
 - Inversement, certains problèmes de plus court chemin peuvent être résolus par programmation dynamique.

Equation de Bellman et plus courts chemins

- ▶ *Equation de Bellman* : formule séquentielle du type
 $f(x) = \min_{y \in p(x)} c(x, y) + f(y)$ avec x, y vecteurs état (multi-dimensionnels) et avec conditions au bord $f(k) = c_k$ pour certains états
 - ▶ **Objectif** : pouvoir calculer toutes les valeurs de $f(x)$ pour les états qui nous intéressent
- ▶ Si l'ensemble des états \mathcal{X} est fini, on peut identifier \mathcal{X} avec $\{1, \dots, n\}$ et la relation s'écrit :

$$f(i) = \min_{j \in p(i)} c(i, j) + f(j)$$

- ▶ On peut ajouter un état artificiel t et poser $f(k) = c(k, t) + f(t)$ avec $c(k, t) = c_k$ et $f(t) = 0$

Plus court chemin et sous-structure optimale

- ▶ Si $u \neq v$, un plus court chemin de u à v doit contenir un nœud $w \neq u$
- ▶ Sous-structure optimale : si $PCC(u, v)$ *existe* et passe par w , alors la partie de $PCC(u, v)$ entre u et w est un PCC
 - ▶ Démonstration par l'absurde :
si ce n'était pas un PCC, on le remplacerait par un PCC
 - ▶ Idem entre w et v
- ▶ NB : Pour garantir l'existence de plus courts chemins, on doit supposer par exemple, que le graphe est sans circuit absorbant. Dans ce cas, il existe toujours un plus court chemin ... simple ! (un chemin peut toujours se décomposer en un chemin simple + des circuits).



Un plus court chemin n'existe pas toujours!

- Contre-exemple (tous les poids des arcs valent -1) :
$$\begin{array}{ccccc} & u & \leftrightarrow & w & \\ & \downarrow & & & \downarrow \\ t & \leftrightarrow & v \end{array}$$
 - Le PCC simple de u à v passe par w , mais le chemin $\{u, w\}$ n'est pas un PCC simple de u à w (le PCC simple de u à w passe par t puis v !)
 - NB : Dans ce graphe, il n'existe pas de plus court chemin (non simple) entre les sommets puisqu'on peut boucler à l'infini.

Chap 2. Calcul d'une solution optimale par prog. dynamique

Plan

1. Exemple du sac à dos binaire

1. Rappel version avec mémoïsation
2. Remontée d'une solution optimale

2. Prog Dyn: la Méthode générale

3. Lien avec les plus courts chemins

4. Conclusion

5. Ex (hors cours) : cageots de fraises

6. TD: rendu monnaie optimal + jeu télévisé

Ce qu'on a vu aujourd'hui

- ◆ Programmation dynamique = calcul récursif de la valeur optimale d'un problème d'optimisation discrète
 - appels top-down, mais calcul **bottom-up!**
- ◆ efficace grâce à la mémoïsation
- ◆ Mémoriser les choix optimaux effectués pour se rappeler des solutions optimales
- ◆ Un autre programme doit être écrit pour construire une solution optimale
 - Calcul **top-down** à partir des choix optimaux (-> coût)
- ◆ *Au final: descente, remontée et redescente*

Chap 2. Calcul d'une solution optimale par prog. dynamique

Plan

1. Exemple du sac à dos binaire

1. Rappel version avec mémoïsation
2. Remontée d'une solution optimale

2. Prog Dyn: la Méthode générale

3. Lien avec les plus courts chemins

4. Conclusion

5. Annexe (hors cours) : cageots de fraises

6. TD: rendu monnaie optimal + jeu télévisé

Enoncé

- ◆ **n** cageots de fraises doivent être distribués dans **k** magasins
- ◆ Les bénéfices que l'on peut retirer de chaque magasin dépendent du nombre de cageot de fraises fourni
- ◆ Comment répartir les cageots entre les magasins pour maximiser les gains?

Démarche

1. Comprendre le problème!
2. Réussir à le **modéliser** correctement
 1. Que cherche-t-on?
 2. Paramètres?
 3. ...
3. Résolution

Exemple

Ex: 3 magasins, 7 cageots

n_j	$b_1(n_j)$	$b_2(n_j)$	$b_3(n_j)$
0	0	0	0
1	3	6	5
2	7	12	10
3	12	16	15
4	17	20	20
5	26	22	25
6	35	24	30
7	45	26	35

$b_j(n_j)$ = bénéfice de la vente de n_j cageots dans le $j^{\text{ème}}$ magasin

Fonctions non linéaires!

Trouver les n_j $j=1..k$

qui maximisent :

$$B = \sum b_j(n_j)$$

avec $\sum n_j = n$

Exemple

$n=3, B_{\max} = ?$

n_j	$b_1(n_j)$	$b_2(n_j)$	$b_3(n_j)$
0	0	0	0
1	3	6	5
2	7	12	10
3	12	16	15
4	17	20	20
5	26	22	25
6	35	24	30
7	45	26	35

$n=5, B_{\max} = ?$

n_j	$b_1(n_j)$	$b_2(n_j)$	$b_3(n_j)$
0	0	0	0
1	3	6	5
2	7	12	10
3	12	16	15
4	17	20	20
5	26	22	25
6	35	24	30
7	45	26	35

$n=7, B_{\max} = ?$

n_j	$b_1(n_j)$	$b_2(n_j)$	$b_3(n_j)$
0	0	0	0
1	3	6	5
2	7	12	10
3	12	16	15
4	17	20	20
5	26	22	25
6	35	24	30
7	45	26	35

Exemple

$n=3, B_{\max} = 17$

n_j	$b_1(n_j)$	$b_2(n_j)$	$b_3(n_j)$
0	0	0	0
1	3	6	5
2	7	12	10
3	12	16	15
4	17	20	20
5	26	22	25
6	35	24	30
7	45	26	35

$n=5, B_{\max} = 27$

n_j	$b_1(n_j)$	$b_2(n_j)$	$b_3(n_j)$
0	0	0	0
1	3	6	5
2	7	12	10
3	12	16	15
4	17	20	20
5	26	22	25
6	35	24	30
7	45	26	35

$n=7, B_{\max} = 45$

n_j	$b_1(n_j)$	$b_2(n_j)$	$b_3(n_j)$
0	0	0	0
1	3	6	5
2	7	12	10
3	12	16	15
4	17	20	20
5	26	22	25
6	35	24	30
7	45	26	35

Gains marginaux

- ◆ On s'intéresse en fait aux gains marginaux: gains supplémentaires obtenus par cageot

$$g_j(p) = b_j(p) - b_j(p-1) \text{ avec } b_j(0) = 0$$

n_j	$b_1(n_j)$	$b_2(n_j)$	$b_3(n_j)$
0	0	0	0
1	3	6	5
2	7	12	10
3	12	16	15
4	17	20	20
5	26	22	25
6	35	24	30
7	45	26	35

+5

n_j	$g_1(n_j)$	$g_2(n_j)$	$g_3(n_j)$
0	0	0	0
1	3	6	5
2	4	6	5
3	5	4	5
4	5	4	5
5	9	2	5
6	9	2	5
7	10	2	5

Question 1

- ◆ Proposer un algorithme optimal dans le cas où la fonction de gains marginaux **g_j est décroissante** pour chaque magasin
- ◆ Donner un contre-exemple si les gains marginaux ne sont pas décroissants

Réponse

◆ Qu. 1: gains marginaux décroissants

- **Algorithme glouton:**

Choix optimal localement (à chaque étape) dans l'espoir d'atteindre un optimum global à la fin

- Ici: chaque cageot est distribué au magasin où il rapporte le plus (compte tenu du nombre de cageots déjà possédés)

Question 2

- ◆ Combien de solutions au problème existent, dans le cas général?

Réponse

◆ Qu. 2: nombre de possibilités pour répartir les n cageots dans k magasins?

- Pour faire k paquets, il faut placer k-1 séparations entre les n cageots

$$\square \mid \square \quad \square \mid \square \quad \square \quad \square \quad n=7, k=3$$

- En tout n+k-1 éléments (indifférenciés)
- Il faut en choisir k-1, qui seront les séparations

$$C_{n+k-1}^{k-1} = \frac{(n+k-1)!}{n!(k-1)!}$$

Question 3

- ◆ Proposer un algorithme de type programmation dynamique qui calcule la répartition optimale des cageots de fraises, dans le cas général

(le plus difficile est de trouver la récurrence!)

Réurrence

◆ Équation de récurrence ?

- Sur les cageots (n)?
Difficile car tout peut changer en ajoutant/enlevant un seul cageot des magasins (voir l'exemple)
- On regarde donc sur les magasins

Réurrence

- On suppose connus les $B_q(p)$ bénéfice maximal pour p cageots ($p=0..n$) dans les k premiers magasins
- Idée : comparer ces bénéfices à ceux obtenus en enlevant des cageots des q premiers magasins pour les mettre dans le q+1^{ème}
- À faire : formaliser cette équation de récurrence

Réurrence

- On suppose connus les $B_q(p)$ bénéfice maximal pour p cageots ($p=0..n$) dans les k premiers magasins
- Idée : comparer ces bénéfices à ceux obtenus en enlevant des cageots des q premiers magasins pour les mettre dans le q+1^{ème}

$$\begin{cases} B_{q+1}(p) = \max_{n_j=0..p} [B_q(p - n_j) + b_{q+1}(n_j)] \\ B_1(p) = b_1(p) \end{cases}$$

Programme récursif

$$\begin{cases} B_{q+1}(p) = \max_{n_j=0..p} [B_q(p - n_j) + b_{q+1}(n_j)] \\ B_1(p) = b_1(p) \end{cases}$$

Double benefMaxProgramme valeurOpt
récursif avec mémoïsation

- ◆ Mémorisation choix optimaux (choixOpt)
- ◆ Programme de construction d'une solution optimale à partir du tableau choixOpt

$$\begin{cases} B_{q+1}(p) = \max_{n_j=0..p} [B_q(p - n_j) + b_{q+1}(n_j)] \\ B_1(p) = b_1(p) \end{cases}$$

Double **benefMax** (int Qinit, int Pinit) {
 double memVal[Qinit, Pinit + 1] ; // déclaration tableau temporaire
 int choixOpt[Qinit,Pinit + 1] ; // *choixOpt[q,p]=n ssi mettre n paniers dans q est optimal quand il en reste p !*

```

double bMax( int q, int p) {
    | if (memVal[q, p] ≠ -1) return memVal[q, p] ;
    | if (q==0) { res = B[0,p] ; choix = p; }
    | else {
    |   res = B[ q, p ] ; choix = p ; // nbre opt de paniers dans q
    |   for (n=0; n < p ; --n) { // n paniers dans le magasin q...
    |     aux = bMax( q-1, p – n ) + B[q, n ];
    |     if (aux > res) { res = aux ; choix = n; }
    |   }
    |   memVal[q, p] = res ;
    |   choixOpt[q, p] = choix ;
    |   return res;
}
// main
return bMax( Qinit – 1 , Pinit) ;
}
```

Construction d'une solution optimale (*top-down*, à partir du tableau *choixOpt*)

```
Int nbPaniersOpt[ Qinit ] ; // une solution optimale

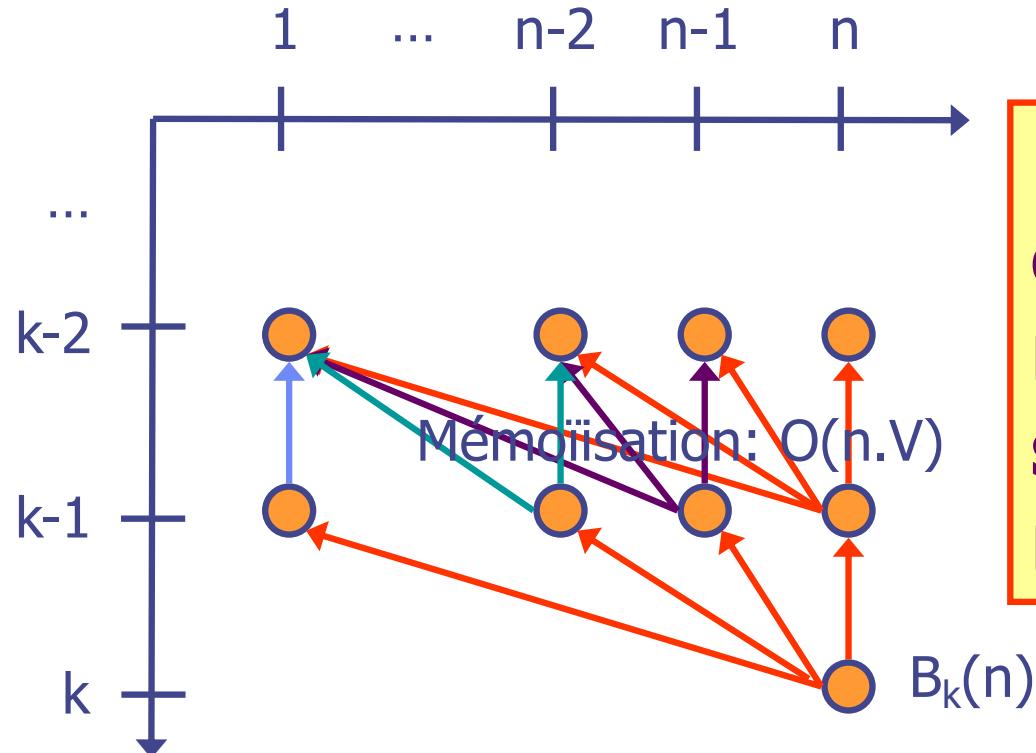
for (q=Qinit-1, p=Pinit; q≥0; --q ) {
    n = choixOpt[q, p ] ;
    nbPaniersOpt[ q ] = n ;
    p = p - n ;
}

// Affichage de la solution optimale calculée
For (q=0; q<Qinit; ++q)
    writeln 'Nombre de paniers dans' + q + 'est' + p ;
```

Exercice

- ◆ dessiner le graphe des appels correspondant au calcul de $B_k(n)$
- ◆ analyser le coût en calcul et en mémoire

Graphe des appels



**Beaucoup de
calculs
redondants
sans la
mémoïsation!**

$$\begin{cases} B_{q+1}(p) = \max_{n_j=0..n} [B_q(p - n_j) + b_{q+1}(n_j)] \\ B_1(p) = b_1(p) \end{cases}$$

Algorithme récursif

Quelles sont les complexités (temps et mémoire) de l'algorithme récursif avec marquage?

Analyse du coût en 3 étapes

1. $C_1 = \#\text{appels récursifs non déjà calculés}$

- Ici: tous les appels $\text{Bmax}(q,p)$ avec $0 \leq q < Q$ et $0 \leq p \leq P$
 $C_1(Q,P) = \sum_{q=0..M-1} \sum_{p=0..P} 1 \text{ appel} = Q.(P+1) \text{ appels}$

2. $C_2 = \#\text{appels « déjà calculés »}$ (i.e. *return mémoïsation*)
= #appels récursifs par appel non calculé

- Ici: $\text{Bmax}(q,p)$ fait les appels récursifs $\text{Bmax}(q-1,k)$ pour $0 \leq k < p$
 $C_2(Q,P) = \sum_{q=0..Q-1} \sum_{p=0..P} p \text{ appels} = Q.P.(P+1) / 2 = \Theta(P^2Q)$

3. $C_3 = \text{coût total des appels non déjà calculés}$

- Ici: $\text{Bmax}(q,p)$ coûte $\Theta(p)$, d'où $C_3(Q,P) = \sum_{q=0..M-1} \sum_{p=0..P} \Theta(p) = \Theta(P^2Q)$

4. Coût total = $C_2 \cdot \Theta(1) + C_3$

- Ici: Coût total = $\Theta(P^2Q)$

Analyse du coût (récursif avec mémoïsation)

1. q magasins, p paniers

Complexité ?

- $O(q.p)$ éléments à calculer (memVal)
 - $O(p)$ pour le calcul du max
 $=> \mathbf{O(p^2.q)}$
-
- ◆ En mémoire : **2 tableaux p.Q ...**

Algorithme itératif

- ◆ À faire : réfléchir à l'algorithme itératif
- ◆ Donner l'algorithme correspondant
- ◆ Quelles sont les complexités de cet algorithme?

Algorithme itératif

2. Solution itérative

- ◆ données nécessaires?
la ligne q ne sert que pour calculer la ligne $q+1 \Rightarrow$ on peut ne conserver qu'une ligne
- ◆ $B_{q+1}(p)$ dépend de $B_q(p)$, $B_q(p-1)$, ... $B_q(1)$
 \Rightarrow on calcule $B_{q+1}(p)$, puis on stocke le résultat dans la ligne à l'indice p
($B_q(p)$ est écrasé, mais il ne sert plus pour calculer les autres éléments de B_{q+1})

Algorithme itératif

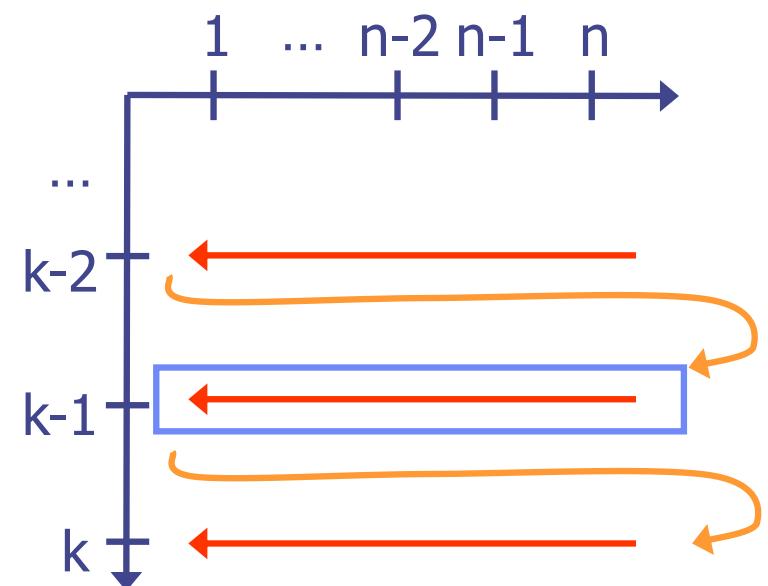
2. Solution itérative

Au final le parcours itératif se fait selon

1. $q = 1 \dots k$ croissant
2. $p = n \dots 1$ décroissant

Complexité ?

Toujours **$O(n^2k)$** en temps
Mais **$O(n)$** en mémoire



Algorithme itératif

Procédure cageots

Pour p de 1 à n Faire

$B[p] = b(1, p)$

 -- init avec les bénéfices du 1^{er} magasin

 Pour q de 2 à k Faire

 -- boucle: q magasins

 Pour p de n à 1 pas -1 Faire

 -- nb de cageots

$B[p] = \max B(q, p)$

Fonction maxB (q, p : entier)

 entier m = B[p]

 -- init avec bénéfices q-1 magasins, p cageots

 Pour j de 1 à p Faire

$m = \max (m, B[p-j] + b(q, j))$

 Retourner m

Annexes

- ◆ TD / Exercice à la maison : jeu télévisé
- ◆ TD / Calcul d'un rendu de monnaie optimal par programmation dynamique
- ◆ Gestion de tableaux multidimensionnels
(vus comme tableaux de tableaux)

Exercice à la maison: Jeu télévisé

- ◆ Roue graduée de 0 à 1 million (tirages uniformes)
Après chaque lancer, on peut dire:
 - ◆ soit STOP : on gagne le montant indiqué en € ;
 - ◆ soit ENCORE: on relance la roue.
- ◆ On peut lancer la roue 4 fois au maximum.
Au 4^{ème} lancer, on gagne la somme indiquée.
- ◆ Ma stratégie: je relance ssi l'espérance de mon gain en relançant est supérieure à mon gain actuel.
- ◆ Au 1^{er} lancer, j'obtiens 700 000.
Quel est mon choix: STOP ou ENCORE ?

Par programmation dynamique

◆ Formulation récursive:

- R_k = tirage au pas k (considéré uniforme dans $[0,1]$)
- G_k = espérance du gain au lancer k
 - ◆ (en Million d'€, donc dans $[0,1]$)
- Avec n lancers au maximum: $G_n=0.5$

◆ Formule récursive:

$$\begin{aligned} G_k &= E[R_k | R_k \geq G_{k+1}] \cdot \Pr(R_k \geq G_{k+1}) + G_{k+1} \cdot \Pr(R_k \leq G_{k+1}) \\ &= (1+G_{k+1})/2 \cdot (1 - G_{k+1}) + G_{k+1} \cdot G_{k+1} \\ &= (1+G_{k+1}^2)/2 \end{aligned}$$

Jeu télévisé

```
#define UNDEFINED_VAL -1 /* Valeur de gain impossible */
double jeuTele(int Nmax) {
    double memVal[Nmax] ; // déclaration tableau temporaire
    double esperanceGain( int k) {
        | double res;
        | if (memVal[k] ≠ UNDEFINED_VAL) return memVal[k] ;
        | if (k==Nmax) { res = 0.5 ; }
        | else {
        |     double tmp = esperanceGain(k+1) ;
        |     res = (1 + tmp*tmp) / 2 ;
        | }
        | memVal[k] = res ; | return res;
    }
}

// main
memVal = allocate_and_init(Nmax, UNDEFINED) ;
double gain = lancer() ; // Premier lancer
for (k=1; k<Nmax; k++)
    if (esperanceGain(k) > gain) gain = lancer() ; // ENCORE : on relance
    else break ; // STOP :
}
#endif
```

Questions: quel est le coût? Que serait-il sans mémoisation?

One more thing...

Tableaux multidimensionnels

- ◆ La mémoire est « unidimensionnelle.
 - ◆ Comment les langages (compilateurs) gèrent-ils les tableaux multidimensionnels ?

Int tab[D1][D2]

Ex: D1=2, D2=3

0	1	2
3	4	5

- ◆ Stockage *Row-Major Order* [0, 1, 2, 3, 4, 5]
(C/C++, python, mathematica, ...) tab[i][j] = (int*)tab[j + i*D2]
- ◆ Stockage *Column-Major Order* [0, 3, 1, 4, 2, 5]
(Fortran, OpenGL, Matlab, S-Plus, Scilab, ...) tab[i][j] = (int*)tab[i + j*D1]
- ◆ Rem. Java: tableau de tableau : [0,1,2] et autre part [3, 4, 5]

One more thing...

Tableaux multidimensionnels

- ◆ Multiplications cachées...
- ◆ Préférer :
 - ◆ tableau unidimensionnel (multiplication explicite)
 $X^* \text{ tab} = (X^*) \text{ calloc}(D1 * D2, \text{sizeof}(X)) ;$
Accès à $\text{tab}[i,j]$ par : $\text{tab}[i*D2 + j]$
 - ◆ Ou tableau de tableau (pas de multiplication mais addressage indirect)
 $X^{**} \text{ tab} = (X^{**}) \text{ calloc}(D1, \text{sizeof}(X)) ;$
 $\text{for (int } i=0; i < D1; ++i) \text{ tab}[i] = (X^*) \text{ calloc}(D2, \text{sizeof}(X)) ;$
Accès à $\text{tab}[i,j]$ par : $\text{tab}[i][j]$ égal à $*(*(\text{tab} + i) + j)$
- **Exercice:** programmer sacMax en C sans multiplication. 71

FIN