UFR d'Informatique

université PARIS DIDEROT PARIS 7

Programmation dynamique

2015--2016

F. Laroussinie

Programmation dynamique

Diviser-pour-régner:

1) on **décompose** un problème en sous-problèmes ***pertinents*** (dont la taille est une fraction de celle de départ),
2) on **résout** les sous-problèmes,
3) on **construit** une solution pour le problème initial…

Approche "top-down"

Programmation dynamique

3 idées !

1) on **résout** des sous-problèmes et on **stocke** leurs solutions,
2) on voit **quels** sous-problèmes sont pertinents et
3) on les utilise pour **construire** une solution pour le problème initial…

Approche "bottom-up"

Exemple : la suite de Fibonacci

$F_0=0 \qquad F_1=1 \qquad F_{n+2} = F_{n+1} + F_n$

```
Def fibonaif(n) :
  if n==0 : return 0
  elif n==1 : return 1
  else: return fibonaif(n-1)+fibonaif(n-2)
```

“top-down”

Exemple : la suite de Fibonacci

$F_0=0 \qquad F_1=1 \qquad F_{n+2} = F_{n+1} + F_n$

```
Def fibocache(n) :
  if n==0 : return 0
  elif n==1 : return 1
  T = [None] * (n+1)
  T[0]=0
  T[1]=1
  return fibocacheAux(T,n)

Def fibocacheAux(T,n) :
  if T[n]==None :
          T[n]=fibocacheAux(T,n-1)+fibocacheAux(T,n-2)
  return T[n]
```

toujours “top-down” (et beaucoup mieux !)

Exemple : la suite de Fibonacci

$F_0=0 \qquad F_1=1 \qquad F_{n+2} = F_{n+1} + F_n$

```
Def fibo(n) :
  T = [0]*(n+1)
  T[1]=1
  for i = 2 … n :
      T[i] = T[i-1]+T[i-2]
  return T[n]
```

“bottom-up”

(2 idées sur les 3…)

Exemple : la suite de Fibonacci

$F_0=0 \qquad F_1=1 \qquad F_{n+2} = F_{n+1} + F_n$

```
Def fiboopt(n) :
  if n==0 : return 0
  elif n==1 : return 1
  a , b = 0 , 1
  for i = 2 … n-1 :
      a , b = b , a+b
  return a+b
```

(évite le stockage des valeurs inutiles pour la suite)

Exemple : la plus grande sous-séquence croissante

Le problème:
Input:
Output: une plus longue sous-séquence croissante

sous-séquence = suite d'indices $1 \leq i_1 < i_2 < \ldots < i_k \leq n$

… croissante : si $a_{i1} < a_{i2} < \ldots < a_{ik}$

Exemple:	5	2	8	6	3	6	9	7
		X			X	X	X	

Exemple : la plus grande sous-séquence croissante

Le problème:
Input:
Output: une plus longue sous-séquence croissante

indice: Soit L[j] la longueur d'une sous-séquence croissante maximale terminant en j.

$$L[j] = 1 + \max(\{0\} \cup \{ L[i] \mid i<j \wedge a_i < a_j \})$$

Longueur de la solution = $\max(\{L[i] \mid 1 \leq i \leq n\})$

Exemple : la plus grande sous-séquence croissante

Exemple: **5 2 8 6 3 6 9 7**

L[j]: 1 1 2 2 2 3 4 4

L[1]=1	5 2 8 6 3 6 9 7
L[2]=1	5 2 8 6 3 6 9 7
L[3]=2	5 2 8 6 3 6 9 7
L[4]=2	5 2 8 6 3 6 9 7
L[5]=2	5 2 8 6 3 6 9 7
L[6]=3	5 2 8 6 3 6 9 7
L[7]=4	5 2 8 6 3 6 9 7
L[8]=4	5 2 8 6 3 6 9 7

Les plus grandes sous-séquences croissantes sont de taille 4.

Exemple : la plus grande sous-séquence croissante

$$L[j] = 1 + \max(\{0\} \cup \{ L[i] \mid i<j \wedge a_i < a_j \})$$

Complexité en $O(n^2)$

Algorithme:

1) calcul des L[j] avec:
   - mémorisation du L[-] max (et du j_{max})
   - pour chaque L[j], on stocke dans Pred[j] un indice i tel que L[j]=1+L[i].
2) On retrouve les L[j_{max}] éléments d'une sous-séquence croissante maximale avec Pred[-]:

j_{max}, Pred[j_{max}], Pred[Pred[j_{max}]], …

Exemple : la plus grande sous-séquence croissante

```
def plssc(T) :
   |T|=n
   maxsofar , imax  = 0 , 0
   L, Pred = [0,...0], [0,...,0]      taille → n
   for j = 0 ... n-1 :
      aux = 0
      iaux = None
      for i in 0 ... j-1 :
         if (T[i] < T[j] and L[i]>aux) :
            aux = L[i]
            iaux = i
      L[j] =1 + aux
      Pred[j] = iaux
      if (L[j] > maxsofar) :
         maxsofar = L[j]
         jmax = j
#Construction de la ss-seq trouvee:
   iaux = jmax
   res = [0...0]      taille → maxsofar
   i = maxsofar-1
   while (iaux != None) :
      res[i] = iaux
      iaux = Pred[iaux]
      i = i-1
   return maxsofar, res
```

Calcul de L[j]

MaJ de max et j_{max}

Construction d'une solution

Exemple : la plus grande sous-séquence croissante

Python

```
def plssc(T) :
   maxsofar , imax  = 0 , 0
   L, Pred = [0]*len(T), [0]*len(T)
   for j in range(len(T)) :
      aux = 0
      iaux = None
      for i in range(j) :
         if (T[i] < T[j] and L[i]>aux) :
            aux = L[i]
            iaux = i
      L[j] = 1 + aux
      Pred[j] = iaux
      if (L[j] > maxsofar) :
         maxsofar = L[j]
         jmax = j

#Construction de la ss-seq trouvee:
   iaux = jmax
   res = [0] * maxsofar
   i = maxsofar-1
   while (iaux != None) :
      res[i] = iaux
      iaux = Pred[iaux]
      i=i-1
   return maxsofar, res
```

Calcul de L[j]

MaJ de max et j_{max}

Construction d'une solution

Les problèmes de sac à dos

Les problèmes de sac à dos

Le problème:

Input: un poids maximal $W \in \mathbf{N}$, un ensemble de n objets ayant chacun un poids

Output: la valeur maximale pouvant être stockée dans le sac (sans dépasser sa capacité !).

Taille du problème: les 2n+1 valeurs données

On distingue des variantes… (et des cas particuliers):

- avec répétitions : on peut prendre plusieurs fois le même objet.
- sans répétition: chaque objet est pris au plus une fois.
- …

Les problèmes de sac à dos

Le problème:
Input: un poids maximal $W \in \mathbf{N}$, un ensemble … hacun un poids
Output: la valeur m … ans dépasser sa capacité !).

Taille du probl… 2n+1 valeurs données

Problèmes connus et difficiles !

- Cas avec répétitions: “Integer knapsack”
- Cas sans répétition: “Knapsack”
- Cas “borné”
- …

Les *problèmes de décision associés* sont NP-complets… (Voir le “Garey & Johnson”, page 247).

sac à dos avec répétitions

Le problème:
Input: un poids maximal $W \in \mathbf{N}$, un ensemble de n sortes d'objets ayant un poids
Output: la valeur maximale pouvant être stockée dans le sac (sans dépasser sa capacité !).

Exemple:

O:	1	2	3	4	5
w_i	2	1	5	6	7
v_i	6	1	18	22	24

W=12

Essais:
- O_5+O_3 -> poids 12, valeur 42
- 2 x O_3 + O_1 -> poids 12, valeur 42
- 2x O_4 -> poids 12, valeur 44,
- …

sac à dos avec répétitions

Input:

Output: la valeur maximale pouvant être stockée dans le sac.

Au travail !

soit **K[w]** la valeur maximale pouvant être stockée dans un sac de capacité $w \in \{0,1,2,\ldots,W\}$

- solution au problème: **K[W]**
- **K[0]=0**

Comment calculer K[w] ?

sac à dos avec répétitions

Calcul de K[w]…

Si une solution optimale pour réaliser K[w] contient au moins un objet i, alors…

1) sans cette occurrence de l'objet i, le contenu du sac est optimal pour le poids $w\text{-}w_i$, et
2) sa valeur est alors K[w-wi] !

$$K[w] = \max(\{0\} \cup \{K[w-w_i]+v_i \mid w_i \leq w\})$$

sac à dos avec répétitions

```
def SaDrepet(W,T) :
    K = [None, … None]    // taille → W+1
    K[0] = 0
    for w = 1 … W :
        M = 0
        for (wi,vi) in T :
            if wi <= w :
                M = max(M,K[w-wi]+vi)
        K[w]=M

    return K[W],K
```

sac à dos avec répétitions

Python

```
def SaDrepet(W,T) :
    K = [None] * (W+1)
    K[0] = 0
    for w in range(1,W+1) :
        M = 0
        for (wi,vi) in T :
            if wi <= w :
                M = max(M,K[w-wi]+vi)
        K[w]=M
    return K[W],K
```

sac à dos avec répétitions

```
def SaDrepet(W,T) :
    K = [None, … None]    // taille → W+1
    K[0] = 0
    for w = 1 … W :
        M = 0
        for (wi,vi) in T :
            if wi <= w :
                M = max(M,K[w-wi]+vi)
        K[w]=M

    return K[W],K
```

Exemple

O:	1	2	3	4	5
w_i	2	1	5	6	7
v_i	6	1	18	22	24

	0	1	2	3	4	5	6	7	8	9	10	11	12
K[-]	0	1	6	7	12	18	22	24	28	30	36	40	44

sac à dos avec répétitions complexité

```
def SaDrepet(W,T) :
    K = [None, … None]    // taille → W+1
    K[0] = 0
    for w = 1 … W :
        M = 0
        for (wi,vi) in T :
            if wi <= w :
                M = max(M,K[w-wi]+vi)
        K[w]=M

    return K[W],K
```

Instance:
- une liste T d'objets de taille n,
- un entier W.

Complexité en $\Theta(W.n)$

sac à dos avec répétitions
complexité

On suppose que les $2n+1$ valeurs sont codées en binaire. La taille d'une instance du problème du sac à dos est donc:

$$t = \sum_i \log(w_i+1) + \sum_i \log(v_i+1) + \log(W+1)$$

$w_i, v_i, W > 0$

Une complexité en $\Theta(W.n)$ est donc en $\Theta(2^t)$.

sac à dos avec répétitions

Construire une solution à partir de K[-]:

```
def SolSaDrepet(W,K,T) :
    n = |T|
    S = [0, …  ,0]     taille → n
    v = K[W]
    w = W
    while v>0 :
        for j = 0 … n-1 :
            (wj,vj) = T[j]
            if (wj <= w) and (v-vj == K[w-wj]) :
                                S[j] += 1
                                v -= vj
                                w -= wj
return S
```

une solution S[-] est un tableau d'entiers tq:

$$\sum S[i].v_i = K[W] \text{ et } \sum S[i].w_i \leq W$$

Complexité en O(W.n)

sac à dos avec répétitions

Construire une solution à partir de K[-]:

Python

```
def SolSaDrepet(W,K,T) :
    n = len(T)
    S = [0]*n
    v = K[W]
    w = W
    while v>0 :
        for j in range(n) :
            (wj,vj) = T[j]
            if (wj <= w) and (v-vj == K[w-wj]) :
                            S[j] += 1
                            v -= vj
                            w -= wj
return S
```

sac à dos sans répétition

sac à dos *sans* répétition

Chaque objet est pris au plus une fois.

Le problème: n *objets*, une capacité W…

Calculer K[w] ?

L'idée utilisée précédemment ne marche pas car les problèmes ne sont plus indépendants: le contenu du sac optimal pour le poids w sans l'objet i n'est pas forcément optimal pour le poids $w-w_i$!

Exemple: W=10 et

O:	1	2	3
w_i	5	5	6
v_i	20	2	3

valeur=2 !

K[10] = 22 (objet 1 + objet 2)

K[5] = 20 (objet 1)

Donc (objet 1+objet 2)\ objet 1 n'est pas optimal pour $10-w_1$!

sac à dos *sans* répétition

Il faut affiner les calculs pour tenir compte des éléments potentiellement présents dans le sac (et qui ne peuvent pas être pris une seconde fois !).

Soit $K[j,w]$ la valeur maximale que l'on peut stocker dans un sac de capacité w avec des objets $1,\ldots,j$.

$$K[j,w] = \max\left(K[j-1,w],\ K[j-1,w-w_j]+v_j\right)$$

$$K[0,w] = 0 \quad \text{et} \quad K[j,0]=0$$

Solution = $K[n,W]$: tous les objets sont autorisés et le poids max est W !

sac à dos *sans* répétition

Algorithme de calcul des K[j,w]:

$K[0,w]=0 \quad \forall\ w=0,\ldots,W$

$K[j,0]=0 \quad \forall\ j=0,\ldots,n$

(NB: on peut aussi calculer par colonnes…)

Pour $j=1,\ldots,n$:

  Pour $w = 1,\ldots,W$:

    Si $w \leq w_j$: $K[j,w]=K[j-1,w]$

    Sinon: $K[j,w] = \max(K[j-1,w],K[j-1,w-w_j]+v_j)$

Retourner $K[n,W]$

Complexité en $O(W.n)$

Mémoire en $O(W.n)$

sac à dos *sans répétition*

Exemple:

W=12

O:	1	2	3	4	5
w_i	1	2	5	6	7
v_i	1	6	18	22	24

$j \backslash w$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1	1
2	**0**	1	6	7	7	**7**	7	7	7	7	7	7	7
3	0	1	6	7	7	**18**	19	24	25	25	25	25	25
4	0	1	**6**	7	7	18	22	24	28	**29**	29	40	41
5	0	1	6	7	7	18	22	24	28	**30**	31	40	42

$+v_5$ (=24)

Solution=42 !