

Programmation dynamique

Inspiré de [CLR94] et [Reb05]

Nicolas Delestre

- 1 Introduction
- 2 Programmation dynamique
- 3 Exemple : Problème du sac à dos
- 4 Conclusion



Combinaison 1 / 4

Exercice

Écrire une fonction `cnp`, qui pour deux entiers positifs n et p ($p \leq n$), retourne le nombre de combinaisons de p éléments parmi n .

Pour rappel :

$$\binom{n}{p} = C_n^p = \begin{cases} 1 & \text{si } p = 0 \text{ ou } n = p \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon} \end{cases}$$

Solution

```

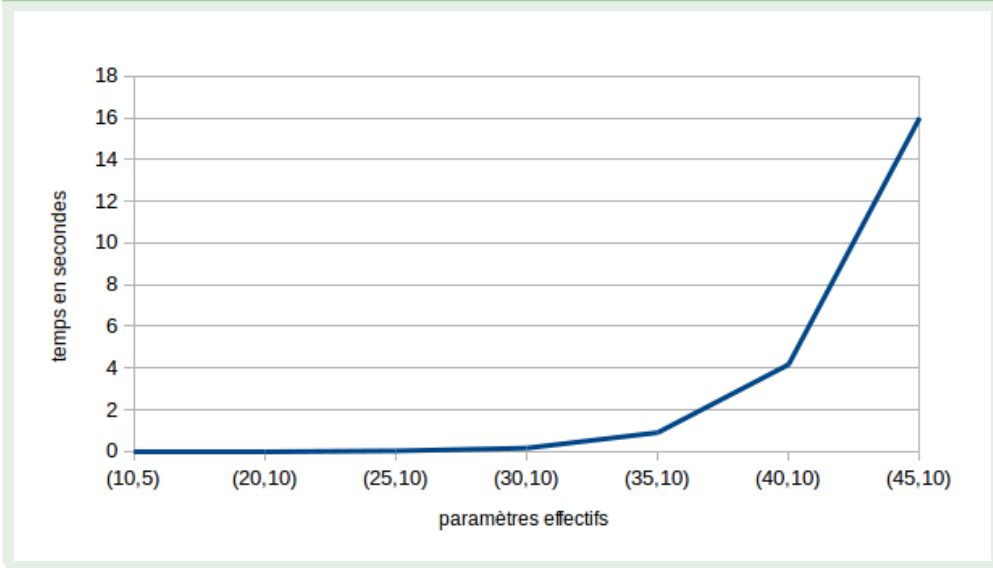
fonction cnp (n,p : naturel) : NaturelNonNul
  |précondition(s)  n ≥ p
debut
  si p=0 ou n=p alors
    retourner 1
  sinon
    retourner cnp(n-1,p)+cnp(n-1,p-1)
  finsi
fin
  
```

Code C

```

1 #include<stdlib.h>
2 #include<stdio.h>
3
4 unsigned int combinaison(unsigned int n,unsigned int p) {
5     if ((p==0) || (n==p))
6         return 1;
7     else
8         return combinaison(n-1,p)+combinaison(n-1,p-1);
9 }
10
11 int main(int argc, char** argv) {
12     unsigned int a,b;
13     if (argc==3) {
14         a = atoi(argv[1]);
15         b = atoi(argv[2]);
16         if (a>=b)
17             printf ("%d\n",combinaison(a,b));
18     }
19     return EXIT_SUCCESS;
20 }
  
```

Temps d'exécution



Prog. Dyn. v1.3.3

Programmation dynamique

Programmation dynamique 1 / 2

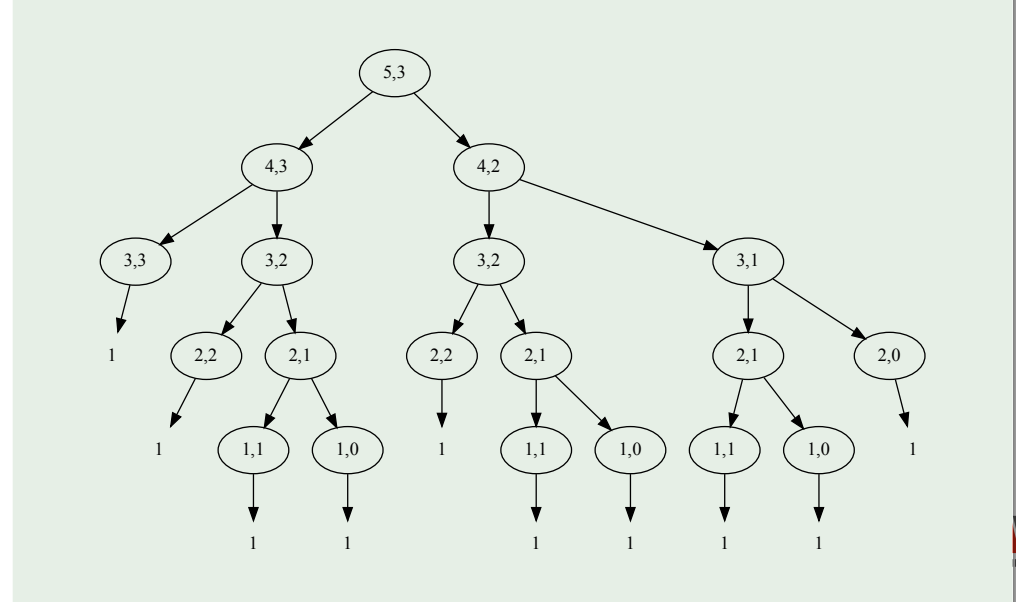
Première définition

« Paradigme de conception qu'il est possible de voir comme une amélioration ou adaptation de la méthode diviser-régner [dans le sens où une] solution d'un problème dépend des solutions précédentes obtenues des sous-problèmes » [Reb05]

Deuxième définition

« La programmation dynamique est une technique algorithmique pour optimiser des sommes de fonctions monotones croissantes sous contrainte. » (Wikipédia)

Arbre des appels



5 / 30 Prog. Dyn. v1.3.3

Programmation dynamique

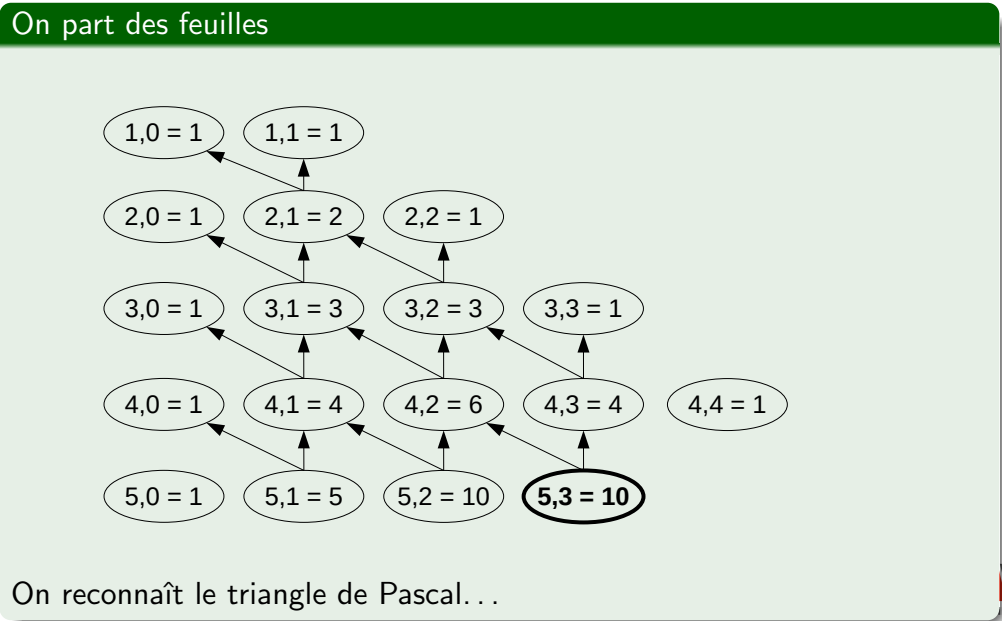
Programmation dynamique 2 / 2

Historique

- Principe invité en 1940 par Richard Bellman
- Pour la petite histoire, Bellman a choisi le terme programmation dynamique dans un souci de communication : son supérieur ne supportait ni le mot « recherche » ni celui de « mathématique ». Alors il lui a semblé que les termes « programmation » [(au sens de la planification)] et « dynamique » donnaient une apparence qui plairait à son supérieur

Amélioration de la méthode « diviser-régner »

- Il faut (tenter d') utiliser la programmation dynamique lorsque les sous-problèmes ne sont pas indépendants et/ou la complexité de l'algorithme « naïf » est élevée (non polynomiale : exponentielle, factorielle)
- C'est un algorithme itératif qui calcule tout d'abord les résultats de base pour les assembler et ainsi calculer le résultat recherché
- Utilisation d'un tableau pour stocker des valeurs calculées qui pourront être réutilisées



En utilisant un tableau à une dimension [Reb05]

```

fonction cnp (n,p : naturel) : NaturelNonNul
  [précondition(s)  n ≥ p et n ≤ MAX_N et p ≤ MAX_P
  Déclaration  i,j : Naturel
                b : Tableau[0..MAX_N] de NaturelNonNul
  debut
    b[0] ← 1
    pour i ← 1 à n faire
      b[i] ← 1
      pour j ← i-1 à 1 pas de -1 faire
        b[j] ← b[j] + b[j-1]
      finpour
    finpour
  retourner b[p]
  fin
    
```

Nouvel algorithme pour cnp

```

fonction cnp (n,p : naturel) : NaturelNonNul
  [précondition(s)  n ≥ p et n ≤ MAX_N et p ≤ MAX_P
  Déclaration  i,j : Naturel
                b : Tableau[0..MAX_N][0..MAX_P] de NaturelNonNul
  debut
    pour i ← 0 à n faire
      pour j ← 0 à min(i,p) faire
        si i=j ou i=0 ou j=0 alors
          b[i,j] ← 1
        sinon
          b[i,j] ← b[i-1,j-1] + b[i-1,j]
        fin
      finpour
    finpour
  retourner b[n,p]
  fin
    
```

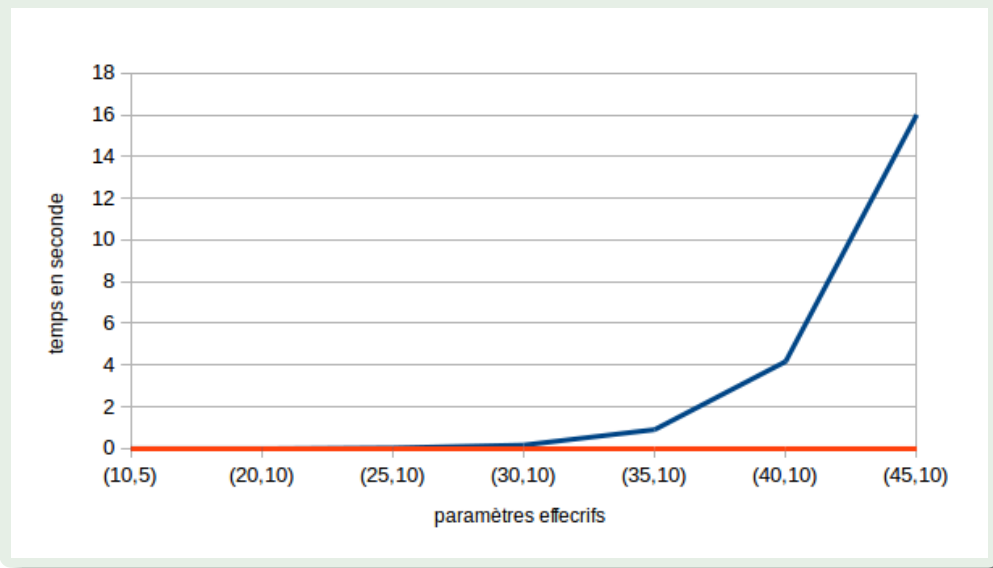
		j					
		0	1	2	3	4	5
i	0	1					
	1	1	1				
	2	1	2	1			
	3	1	3	3	1		
	4	1	4	6	4	1	
5	1	5	10	10	5	1	

Code C

```

1 #include <stdlib.h>
2 #include <stdio.h>
3
4 unsigned int combinaison(unsigned int n, unsigned int p) {
5     int i, j;
6     unsigned int b[n+1];
7     b[0] = 1;
8     for (i=1; i<=n; i++) {
9         b[i] = 1;
10        for (j=i-1; j>0; j--)
11            b[j] = b[j] + b[j-1];
12    }
13    return b[p];
14 }
15
16 int main(int argc, char** argv) {
17     unsigned int a, b;
18     if (argc==3) {
19         a = atoi(argv[1]);
20         b = atoi(argv[2]);
21         if (a>=b)
22             printf("%d\n", combinaison(a,b));
23     }
24     return EXIT_SUCCESS;
25 }
    
```

Temps d'exécution



Présentation du problème

- Soit n objets i d'un certain poids w_i et d'une certaine valeur v_i
- Soit un sac à dos permettant de stocker au maximum des objets dont la somme des poids est W
- Quels objets mettre dans le sac de façon à maximiser la valeur de l'ensemble ?

Présentation mathématique du problème

- Soit $X \in [0, 1]^n$ tel que $x_i = 1$ si l'objet est dans le sac à dos, 0 sinon.
- L'objectif est de choisir X afin de maximiser $\sum_{i=1}^n x_i v_i$ tel que $\sum_{i=1}^n x_i w_i \leq W$



Signature de la fonction

fonction resoudreSacADos (tailleDuSac, nbObjets : **NaturelNonNul** ; valeurs, poids : **Tableau[1..MAX] de Naturel**) : Ensemble<**NaturelNonNul**>
 [précondition(s) nbObjets ≤ MAX

Exemple [Reb05]

- Données du problème :
 - Taille du sac : 11
 - Caractéristiques des objets :

objets	1	2	3	4	5
valeurs	1	6	18	22	28
poids	1	2	5	6	7

- Solution : {3,4} avec la valeur 40

Résolution en force brute

fonction resoudreSacADos (tailleDuSac, nbObjets : **NaturelNonNul** ; valeurs, poids : **Tableau[1..MAX_OBJETS] de Naturel**) : Ensemble<**NaturelNonNul**>
 [précondition(s) nbObjets ≤ MAX_OBJETS
 Déclaration i : **NaturelNonNul**
 objetsDispos : Ensemble<**Naturel**>
debut
 objetsDispos ← ensembleNaturels(1,nbObjets)
retourner resoudreSacADosR(tailleDuSac,ensemble(),objetsDispos,valeurs,poids)
fin

On suppose posséder :

- **fonction** ensembleNaturels (borneInf, borneSup) : Ensemble<**Naturel**>
- **fonction** somme (indice : Ensemble<**Naturel**>, valeurs : **Tableau[1..MAX] de Naturel**) : **Naturel**
- **procédure** obtenirSupprimerUnElement (**E/S** ens : Ensemble<**Naturel**>, **S** el : **Naturel**)

Il faut donc l'algorithme de :

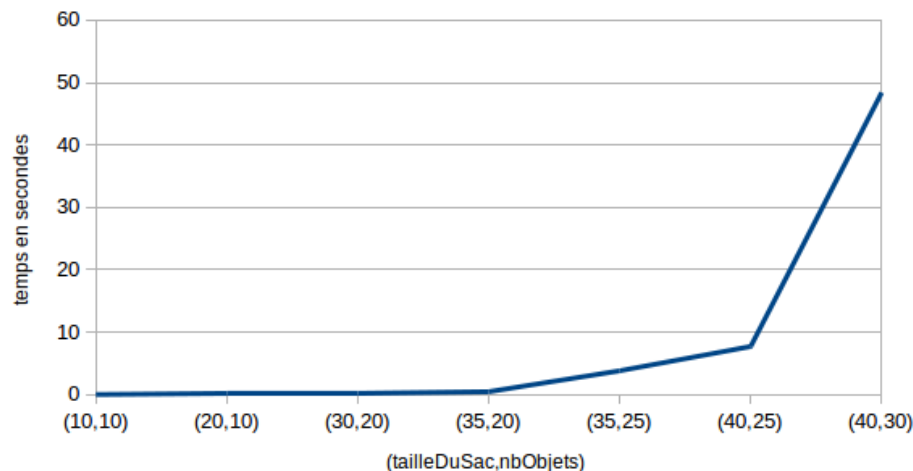
- **fonction** resoudreSacADosR (tailleDuSac : **NaturelNonNul**, objetsChoisis, objetsRestants : Ensemble<**NaturelNonNul**> ; valeurs, poids : **Tableau[1..MAX] de Naturel**) : Ensemble<**NaturelNonNul**>

Résolution en force brute

Trois cas (disjoints) possibles :

- 1 La somme des poids des objets choisis est plus grande que la taille du sac à dos, on retourne alors un ensemble vide
- 2 Il n'y a plus d'objets restants, on retourne alors les objets choisis
- 3 On résout le problème avec un élément el de moins pour les objets restants. On obtient ainsi un ensemble d'objets ens . On retourne l'ensemble ens ou $ens \cup \{el\}$, celui dont la somme des valeurs est la plus grande

Temps d'exécution



Résolution en force brute

fonction resoudreSacADosR (tailleDuSac : **NaturelNonNul**, objetsChoisis, objetsRestants : Ensemble<**NaturelNonNul**>, valeurs, poids : Tableau[1..MAX.OBJETS] de **Naturel**) : Ensemble<**NaturelNonNul**>

Déclaration temp1,temp2,objetsChoisisAvecEl : Ensemble<**NaturelNonNul**>
el : **NaturelNonNul**

debut

si somme(objetsChoisis,poids)>tailleDuSac alors
retourner ensemble()

sinon

si estVide(objetsRestants) alors
retourner objetsChoisis

sinon

obtenirSupprimerUnElement(objetsRestants,el)

objetsChoisisAvecEl ← objetsChoisis

ajouter(objetsChoisisAvecEl,el)

temp1 ← resoudreSacADosR(tailleDuSac,objetsChoisis,objetsRestants,valeurs,poids)

temp2 ← resoudreSacADosR(tailleDuSac,objetsChoisisAvecEl,objetsRestants,valeurs,poids)

si somme(temp1,valeurs)>somme(temp2,valeurs) alors

retourner objetsChoisis

sinon

retourner objetsChoisisAvecEl

finsi

finsi

fin

$O(2^n)$

Programmation dynamique : analyse du problème

- Soit $V_{i,j}$ la somme des valeurs des objets retenus parmi les i premiers objets pour un sac à dos de contenance maximale j . Résoudre le problème du sac à dos pour n objets avec un sac de contenance maximale W revient à calculer $V_{n,W}$. Déterminons une relation récursive de $V_{i,j}$ en s'inspirant d'une démonstration par récurrence :
 - Mettre aucun objet quelque soit le sac donne la valeur 0, donc $V_{0,j} = 0$
 - On ne peut mettre aucun objet dans un sac de taille 0, donc $V_{i,0} = 0$
 - Supposons que $V_{i-1,0..j}$ donne les valeurs maximales recherchées pour les $i-1$ premiers objets pour les sacs à dos de taille $0..j$. Lorsque l'on essaye d'ajouter le i ème objet deux cas se présentent :
 - 1 il est plus lourd que le sac à dos ($w_i > j$), on ne peut donc pas l'ajouter, $V_{i,j} \leftarrow V_{i-1,j}$
 - 2 il est plus léger (ou égal) au sac à dos ($w_i \leq j$). Dans ce cas on ne retient ce i ème objet que si sa valeur additionnée à la valeur d'un sac de poids maximal $j - w_i$ ($v_i + V_{i-1,j-w_i}$) est plus grande que la valeur précédemment calculée pour un sac de même taille avec les $i-1$ objets ($V_{i-1,j}$)

Problème du sac à dos 8 / 15

Exemple [Reb05]

Données du problème :

- Taille du sac : 11
- Caractéristiques des objets :

objets	1	2	3	4	5
valeurs (v_i)	1	6	18	22	28
poids (w_i)	1	2	5	6	7

Calcul de V (ne pas oublier que les $V_{i,j}$ sont sommes de v_k) :

Poids maximal du sac à dos (j)

i	w_i	0	1	2	3	4	5	6	7	8	9	10	11
Objets	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	1	1	1	1	1	1	1	1	1	1
2	2	0	1	6	7	7	7	7	7	7	7	7	7
3	5	0	1	6	7	7	18	19	24	25	25	25	25
4	6	0	1	6	7	7	18	22	24	28	29	29	40
5	7	0	1	6	7	7	18	22	28	29	34	35	40

Cas n°1 : $5 > 3$
 $V_{3,3} \leftarrow V_{2,3}$

Cas n°2 : $5 < 7$
 $j - w_i = 7 - 5 = 2$, on compare donc :
 $V_{2,2} + v_3 = 24$ et $V_{2,7} = 7$ donc $V_{3,7} \leftarrow 24$

Problème du sac à dos 10 / 15

Quels objets retenir ?

- Il faut partir de $V_{n,W}$ ($i \leftarrow n$ et $j \leftarrow W$) et remonter les lignes ($i \leftarrow i - 1$) jusqu'à la première ($i = 1$) ou s'arrêter lorsqu'il n'y a plus de place dans le sac à dos ($j = 0$)
- À chaque itération, deux cas se présentent :
 - Si $V_{i,j} \neq V_{i-1,j}$ alors il faut garder le i ème objet et se positionner sur un sac plus petit (colonne $j - w_i$)
 - Si non, ne pas garder le i ème objet et rester sur la même contenance du sac (même colonne j)

Poids maximal du sac à dos (j)

i	w_i	0	1	2	3	4	5	6	7	8	9	10	11
Objets	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	1	1	1	1	1	1	1	1	1	1
2	2	0	1	6	7	7	7	7	7	7	7	7	7
3	5	0	1	6	7	7	18	19	24	25	25	25	25
4	6	0	1	6	7	7	18	22	24	28	29	29	40
5	7	0	1	6	7	7	18	22	28	29	34	35	40

objets retenus = {4}

Problème du sac à dos 9 / 15

Quels objets retenir ?

- Il faut partir de $V_{n,W}$ ($i \leftarrow n$ et $j \leftarrow W$) et remonter les lignes ($i \leftarrow i - 1$) jusqu'à la première ($i = 1$) ou s'arrêter lorsqu'il n'y a plus de place dans le sac à dos ($j = 0$)
- À chaque itération, deux cas se présentent :
 - Si $V_{i,j} \neq V_{i-1,j}$ alors il faut garder le i ème objet et se positionner sur un sac plus petit (colonne $j - w_i$)
 - Si non, ne pas garder le i ème objet et rester sur la même contenance du sac (même colonne j)

Poids maximal du sac à dos (j)

i	w_i	0	1	2	3	4	5	6	7	8	9	10	11
Objets	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	1	1	1	1	1	1	1	1	1	1
2	2	0	1	6	7	7	7	7	7	7	7	7	7
3	5	0	1	6	7	7	18	19	24	25	25	25	25
4	6	0	1	6	7	7	18	22	24	28	29	29	40
5	7	0	1	6	7	7	18	22	28	29	34	35	40

objets retenus = {}

Problème du sac à dos 11 / 15

Quels objets retenir ?

- Il faut partir de $V_{n,W}$ ($i \leftarrow n$ et $j \leftarrow W$) et remonter les lignes ($i \leftarrow i - 1$) jusqu'à la première ($i = 1$) ou s'arrêter lorsqu'il n'y a plus de place dans le sac à dos ($j = 0$)
- À chaque itération, deux cas se présentent :
 - Si $V_{i,j} \neq V_{i-1,j}$ alors il faut garder le i ème objet et se positionner sur un sac plus petit (colonne $j - w_i$)
 - Si non, ne pas garder le i ème objet et rester sur la même contenance du sac (même colonne j)

Poids maximal du sac à dos (j)

i	w_i	0	1	2	3	4	5	6	7	8	9	10	11
Objets	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	1	1	1	1	1	1	1	1	1	1
2	2	0	1	6	7	7	7	7	7	7	7	7	7
3	5	0	1	6	7	7	18	19	24	25	25	25	25
4	6	0	1	6	7	7	18	22	24	28	29	29	40
5	7	0	1	6	7	7	18	22	28	29	34	35	40

objets retenus = {4,3}

Algorithme

```

fonction toutesLesValeursPossibles (tailleDuSac, nbObjets : NaturelNonNul; valeurs, poids : Tableau[1..MAX_OBJETS] de Naturel) : Tableau[0..MAX_OBJETS][0..TAILLE_MAX_SAC] de Naturel
  [précondition(s)  nbObjets ≤ MAX_OBJETS et tailleDuSac ≤ TAILLE_MAX_SAC
  Déclaration  v : Tableau[0..MAX_OBJETS][0..TAILLE_MAX_SAC] de Naturel
                i, j : Naturel
debut
  pour i ← 0 à nbObjets faire
    v[i,0] ← 0
  finpour
  pour j ← 0 à tailleDuSac faire
    v[0,j] ← 0
  finpour
  pour i ← 1 à nbObjets faire
    pour j ← 1 à tailleDuSac faire
      si poids[i] ≤ j alors
        v[i,j] ← max(v[i-1,j], valeurs[i]+v[i-1,j-poids[i]])
      sinon
        v[i,j] ← v[i-1,j]
      fin
    finpour
  finpour
retourner v
fin

```

Algorithme

```

fonction rechercherObjetsARetenir (tailleDuSac, nbObjets : NaturelNonNul;
                                   v : Tableau[0..MAX_OBJETS][0..TAILLE_MAX_SAC] de Naturel;
                                   poids : Tableau[1..MAX_OBJETS] de Naturel ) : Ensemble<NaturelNonNul>
  [précondition(s)  nbObjets ≤ MAX_OBJETS et tailleDuSac ≤ TAILLE_MAX_SAC
  Déclaration  i, j : Naturel
                resultat : Ensemble<NaturelNonNul>
debut
  resultat ← ensemble()
  i ← nbObjets
  j ← tailleDuSac
  tant que i > 0 et j > 0 faire
    si v[i,j] ≠ v[i-1,j] alors
      ajouter(resultat,i)
      j ← j-poids[i]
    fin
    i ← i-1
  fin tant que
retourner resultat
fin

```

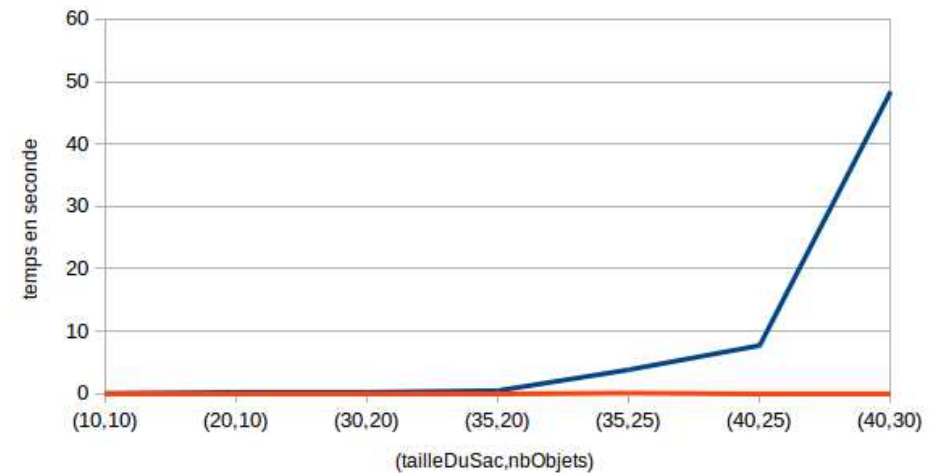
Algorithme

```

fonction résoudreSacADos (tailleDuSac, nbObjets : NaturelNonNul;
                           valeurs, poids : Tableau[1..MAX_OBJETS] de Naturel) : Ensemble<NaturelNonNul>
  [précondition(s)  nbObjets ≤ MAX_OBJETS et tailleDuSac ≤ TAILLE_MAX_SAC
  debut
  retourner rechercherObjetsARetenir(tailleDuSac,
                                       nbObjets,
                                       toutesLesValeursPossibles(tailleDuSac, nbObjets, valeurs, poids),
                                       poids)
  fin

```

Temps d'exécution



- Utiliser la programmation dynamique lorsque :
 - besoin d'une solution exacte
 - il y a une résolution « récursive » du problème mais que l'algorithme « naïf » est d'une grande complexité (car un même calcul est réalisé plusieurs fois)
- Attention : peut être difficile à concevoir
- Certains langages proposent des outils (par exemple des décorateurs) permettant de faciliter le développement de ce type d'algorithme

Remarques

- « Martelli a démontré en 1976 que tout algorithme de programmation dynamique pouvait se ramener à la recherche du plus court chemin dans un graphe. Or, les techniques de recherche heuristique basées sur l'algorithme A* permettent d'exploiter les propriétés spécifiques d'un problème pour gagner en temps de calcul. Autrement dit, il est souvent plus avantageux d'exploiter un A* que d'utiliser la programmation dynamique. » (Wikipédia)
- Lorsque la recherche d'une solution exacte est trop coûteuse, on peut essayer de trouver des solutions approchées :
 - algorithmes glouton
 - algorithme du recuit simulé
 - algorithme tabou
 - etc.

Exemple de problèmes pouvant nécessiter la programmation dynamique

- Distance de Levenshtein : distance entre chaînes de caractères fondée sur le coût de trois opérations (insertion, suppression, substitution)
- Problème de rendu de monnaie
- Problème du voyageur de commerce : plus court chemin qui permet de passer par n sommets d'un graphe ($O(n!) \rightarrow O(n^2 2^n)$ cf. Wikipédia)

n	1	...	8	9	10	11	12	13	...
n!	1	...	40320	362880	3628800	39916800	479001600	6227020800	...
$n^2 \times 2^n$	2	...	16384	41472	102400	247808	589824	1384448	...

- Multiplications chaînées de matrices
 - ex : $|A_1| = (10, 100)$, $|A_2| = (100, 5)$ et $|A_3| = (5, 50)$, Nombre d'opérations pour $(A_1 A_2) A_3$ et $A_1 (A_2 A_3)$?

[CLR94] Thomas Cormen, Charles Leiserson, and Ronald Rivest.

Introduction à l'algorithmique.

Dunod, 1994.

[Reb05] Djamel Rebaïne.

Analyse et conception d'algorithme, chapitre 5.

<http://www.uqac.ca/rebaïne/8INF806/Chapitre5propreprogrammationdynamique.pdf>, 2005.