# Projet de Programmation Nécessaire

Faire son sac

***À chaque fois c'est pareil !***

***Que prendre pour une promenade, une randonnée?***

***Quoi mettre dans son sac-à-dos?***

***Une solution possible est de se dire:***

***Et si j'laissais m'man l'préparer pour moi!?***

***Seulement un jour viendra où elle ne sera plus disponible.***

***Ce jour là, . . .***

***il y aura l'informatique et tout le saint-frusquin. . .***

**Oufff!**

Votre mission, si vous l'acceptez, est de remplir au mieux ce sac à dos.

Pour ce faire, vous aurez des critères qui sont assez évidents:

- Le sac à dos a un volume maximal $V_{\text{max}}$ à ne pas dépasser,
- Les objets susceptibles d'y être rangés sont au nombre de $N$,
- Chaque objet $o_i$ possède:
  - Un volume $v_i$ tel que:

$$\sum_{i=1}^{N} v_i > V_{\text{max}} \qquad \text{et} \qquad v_i \leq V_{\text{max}}\ \forall i \in \{1, \ldots, N\}$$

  - Une utilité $u_i$.

## Définition

`Les objets sont à ranger suivant leurs valeurs d'utilité tout en occupant au plus le sac à dos.`

## Indication

*Si les objets n'ont pas d'indices d'utilité, leurs volumes prend ce rôle :*

$$u_i = v_i$$

## Remarque

*Ne vous fiez pas au apparence !*

*Les algorithmes que vous développerez sont les initiateurs d'autres algorithmes permettant, par exemple, d'ordonner des tâches...*

Vous allez développer 3 algorithmes pour remplir votre SÀD :

- Une version **programmation récursive** de complexité temporelle de $\mathcal{O}(2^N)$,
- Une version **programmation dynamique** à l'aide d'un **tableau des états** de complexité temporelle en $\mathcal{O}(N \times V_{\mathsf{max}})$ mais de complexité en mémoire en $\mathcal{O}(N \times V_{\mathsf{max}})$,
- Une version **programmation dynamique** à l'aide d'une **liste des états potentiels** également de complexité temporelle en $\mathcal{O}(N \times V_{\mathsf{max}})$ mais de complexité mémorielle en $\mathcal{O}(V_{\mathsf{max}})$.
- Vous trouverez sur **Arche – PNé** les fichiers d'entête ainsi que le *code à compléter* des fichiers sources.

L'algorithme principal (`main`) est donné.

*Vous ne pouvez le modifier sans mon accord*

Le reste est à compléter

Comme l'indique les `...`

Et autres `/** TODO **/`.

## *L'algorithme principal (Partie I).*

```c
int main(int argc, char **argv) {
  /** @brief main parameters are :
    * - argc : # of parameters
    * - argv : a vector of string ; each one is a parameter (as a string) */
  if (argc < 4) {
    fprintf(stderr, "USAGE\n\tdp Mode(R|A|L) Utilié(0|1) Vmax vol_1, use_1 ..., vol_n, use_n\n");
    fprintf(stderr, "\t* Mode is\n");
    fprintf(stderr, "\t\t- R for recursive approach,\n");
    fprintf(stderr, "\t\t- A for array approach,\n");
    fprintf(stderr, "\t\t- L for list approach\n");
    fprintf(stderr, "\t* Utilité prend 1 pour des objets avec utilités, 0 sinon\n");
    fprintf(stderr, "\t* Vmax is for bag max volume\n");
    fprintf(stderr, "\t* vol_i is for #i object's volume, i in {1, ... , n}");
    fprintf(stderr, "\t* use_i is for #i object's utility, i in {1, ... , n}");
    exit(-1);
  }
```

# *L'algorithme principal (Partie II).*

```
const char mode = argv[1][0];
const bool utility = (atoi(argv[2]) == 0) ? false : true;
const int Vmax = atoi(argv[3]);
struct retained_t *retained_objects = new_bag();
struct objects_t * object_set = new_objects(argc, argv, utility);
#ifdef _TRACE_
  fprintf(stderr, "mode=%c usefullness=%c Vmax=%d\n", mode, (utility)?'U':'N',
       Vmax);
  fprintf(stderr, "Object set full with %d objects\n", object_set->nb_objects);
  view_objet_set(object_set);
#endif
if(mode == 'R') {
  struct retained_t * my_bag = new_bag();
  prec(Vmax, object_set, my_bag);
  view_bagpack(my_bag, "Final selection");
} else if(mode == 'A') {
  dp_array(Vmax, object_set);
} else {
  dp_list(Vmax, object_set);
}
```

# global.h

*Quelques constantes pour connaître les états du sac-à-dos.*.

```
#ifndef INFTY
#define INFTY -1
#endif

#ifndef UNDTR
#define UNDTR -2
#endif

#ifndef _TRACE_
#define _TRACE_
#endif

#ifndef _DEBUG_
#define _DEBUG_
#endif
```

- Les outils
  - Les objets et l'ensemble des objets
  - Le paquetage

**Remarque**

*À noter que la modélisation du paquetage ne concerne que l'approche récursive. Les approches dynamiques quant à elle embarquent de facto la modélisation du paquetage.*

- Les objets sont modélisés par le TA `object_t` qui caractérise chaque objet par ses volume et utilité.
- L'ensemble de tous les objets *disponibles* est modélisé par le TA `objects_t` qui mémorise les objets dans le vecteur `objects` (de pointeurs) d'objets et qui informe sur leur nombre `nb_objects` ainsi que sur l'indice `first_idx` de celui à traiter en premier.

**Indication**

*Notez que les objets d'un ensemble sont rangés dans un vecteur puisque leur nombre est donné en argument (`argc`).*

## *Les* TA `object_t` *et* `objets_t`.

```
struct object_t {
  int volume;
  int utility;
};

struct objects_t {
  struct object_t * objects;
  int nb_objects;
  int first_idx;
};

struct objects_t * new_objects(int argc, char ** argv, bool utility);
void view_object(struct object_t * object);
void view_objet_set(struct objects_t * set);
```

## *Définition de la fonction* `new_objects`*.*

```
struct objects_t * new_objects(int argc, char ** argv, bool utility) {
    struct objects_t * set = ...;
    int i;
    set->nb_objects = (utility) ? (argc - 3) / 2 : (argc - 3);
    set->objects = ...;
    if(utility) {
        int j;
        for(i = 3, j = 0; i < argc; i += 2, j += 1) {
          /** TODO **/
        }
    } else
        for(i = 3; i < argc; i += 1) {
          /** TODO **/
        }
    return set;
}
```

*Définitions des procédures d'affichage.*

```c
void view_object(struct object_t * object) {
    printf("(%d , %d) ", object->volume, object->utility);
}
void view_objet_set(struct objects_t * set) {
  printf("\n*** View objet set ***\n");
  for(int o = 0; o < set->nb_objects; o += 1) view_object(set->objects + o);
  printf("\n*********************\n");
}
```

Le TA `retained_t` modélise un sac-à-doc *éventuel* sous la forme d'un couple composé de:

- La liste `objects_list` de ses objets;
- La somme des utilités `utilities_sum` de ces objets.

Il est muni des fonctions et procédures intrinsèques suivantes:

- `new_bag` construit un sac-à-dos vide ;
- `bagcpy` fait un duplicata d'un sac-à-dos ;
- `free_bag` libère la mémoire occupée par un sac-à-dos ;
- `clean_bag` vide le sac-à-dos, *sans détruire la variable le modélisant* ;
- `push_object_in_bag` range un nouvel objet dans le sac-à-dos ;
- `view_bagpack` visualise les objets rangés dans un sac-à-dos.

*Déclarations du* TA `retained_t`

*des fonctions et des procédures attachées.*

```c
/**
 * abstract type to record objects put in the bag until now
**/
struct retained_t {
  struct list_t * objects_list;
  int utilities_sum;
};

struct retained_t * new_bag();
void bagcpy(struct retained_t * newbagpack, const struct retained_t * bagpack);
void free_bag(struct retained_t * bagpack);
void clean_bag(struct retained_t * bagpack);
void push_object_in_bag(struct retained_t * bagpack, struct object_t * ptr_object);
void view_bagpack(struct retained_t * bagpack, const char * title);
```

# *Définitions des fonctions et procédures.*

```c
struct retained_t * new_bag() {
  struct retained_t * bagpack = ...;
  bagpack->objects_list = ...;
  assert(bagpack->objects_list);
  return bagpack;
}
void bagcpy(struct retained_t * duplicata, const struct retained_t * bagpack) {
  duplicata->objects_list = listcpy(bagpack->objects_list);
  duplicata->utilities_sum = bagpack->utilities_sum;
}
void free_bag(struct retained_t * bagpack) {/** TODO **/}
void clean_bag(struct retained_t * bagpack) {/** TODO **/}
void push_object_in_bag(struct retained_t * bagpack, struct object_t * ptr_object)
    {/** TODO **/}
void view_bagpack(struct retained_t * bagpack, const char * title) {
  void (* ptr_view_fct) (const struct object_t *) = &view_object;
  printf("\n****************\nVIEW BAGPACKAGING\t%s\n", title);
  view_list(bagpack->objects_list, ptr_view_fct);
  printf("\t\tWith utilities sum = %d\n\n", bagpack->utilities_sum);
}
```

## Definition

La programmation récursive (en abrégé `P_rec`) construit *implicitement* l'arbre de toutes les combinaisons *possibles* d'objets pour en retenir la meilleure.

Ainsi sa complexité dans le *pire cas* est la somme de toutes les combinaisons :

$$\sum_{k=0}^{N-1} \binom{N}{k} = 2^N - 1$$

Il s'ensuit que sa complexité temporelle en le pire cas est de $\mathcal{O}\left(2^N\right)$

Le schéma algorithmique est de type PVH.

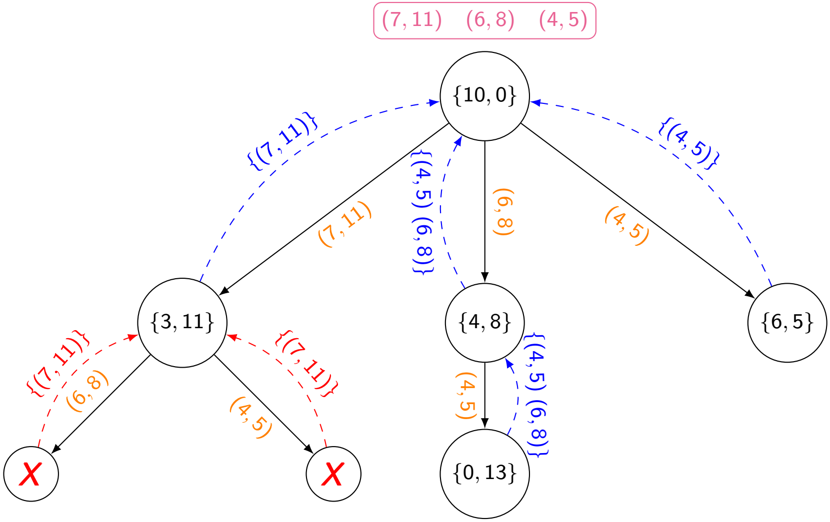
La prédiction est que le sac-à-dos vide est la meilleure configuration, ce qui est évidemment absurde.

La vérification consiste à prendre le sac avec les objets s'y trouvant déjà au début de la procédure, d'y ajouter un nouvel objet et de faire l'appel récursif avec ce nouvel état du sac-à-dos. Au retour de cet appel récursif on met à jour la meilleure configuration si besoin. . .

On opère ce procédé sur chacun des objets *disponibles*, c-à-d. qui ne sont pas déjà rangé dans le sac au début de la procédure. Ainsi à la fin de l'itération sur tous les objets disponibles, la meilleure configuration portera bien son nom !

## *Schéma général de l'approche récursive.*

```
void p_rec(const int Vmax, struct objects_t * obj_set, struct retained_t * bag) {
  const int nb_objects = obj_set->nb_objects;
  struct retained_t * duplicata = ...; bagcpy(...);
  struct retained_t * best_bagpack = ...; bagcpy(...);// Pred: best bag is bag
  for(int obj_idx = obj_set->first_idx ...) {// Verif: Try new objects
    struct object_t * ptr_object = ...;
    int curr_volume = ...;
    if(...) {
      bagcpy(...);
      push_object_in_bag(...);
      obj_set->first_idx = ...;
      p_rec(...);
      if(...) {
        clean_bag(...);
        bagcpy(...);
      }}}
  clean_bag(bagpack);
  bagcpy(bagpack, best_bagpack);
  free_bag(best_bagpack);
}
```


(7, 11) (6, 8) (4, 5)
{10, 0}
{(7, 11)}
(7, 11)
{(4, 5) (6, 8)}
(6, 8)
(4, 5)
{(4, 5)}
{3, 11}
{4, 8}
{6, 5}
{(7, 11)}
(6, 8)
(4, 5)
{(7, 11)}
(4, 5)
{(4, 5) (6, 8)}
X
X
{0, 13}

Soient un sac-à-dos de $V_{\text{max}} = 10$ et 3 objets $(7, 11)$, $(6, 8)$ et $(4, 5)$

L'approche récursive suit les chemins suivants :

**First level recursive call :**

$(7, 11)$ $V = 10$ $[]$ $U = 0$

**Second level recursive call :**

$(6, 8)$ $V = 3$ $\mathbf{[(7, 11)]}$ $U = 11$

**After recursice call :**

Best pack $\mathbf{[(7, 11)]}$ $U = 11$

**Second level recursive call :**

$(4, 5)$ $V = 3$ $\mathbf{[(7, 11)]}$ $U = 11$

**After recursice call :**

Best pack $\mathbf{[(7, 11)]}$ $U = 11$

**Fisrt level recursive call :**

$(6, 8)$, $V = 10$ $[]$ $U = 0$

**Second level recursive call :**

$(4, 5)$ $V = 4$ $\mathbf{[(6, 8)]}$ $U = 8$

**After recursive call :**

Best pack $\mathbf{[(6, 8), (4, 5)]}$ $U = 13$

**First level recursive call :**

$(4, 5)$, $V = 0$ $[]$ $U = 5$

**After recursive call :**

Best pack $\mathbf{[(6, 8), (4, 5)]}$ $U = 13$

En observant attentivement l'approche récursive, on peut construire une approche appelée *programmation dynamique* de complexité temporelle moindre.

Dans une *version statique* de la PD, la complexité mémorielle est (trop) importante. Vous le constaterez en la codant.

Dans sa *version dynamique*, la PD conserve toujours la complexité temporelle souhaitée tout en assurant une complexité mémorielle minimale... *Comme m'man l'aurait fait ; le paradis donc !*

Mais commençons par la version statique pour ensuite la modifier en sa version dynamique.

Pour éviter toute mauvaise interprétation, posons quelques définitions :

**Definition (Objet disponible)**

Un objet disponible est un objet faisant partie des objets de l'ensemble initialement proposé mais qui n'a pas encore été rangé dans le sac-à-dos.

**Definition (Objet valide)**

Un objet valide est un objet disponible dont le volume est au plus égal au *volume libre* du sac-à-dos.

### Definition (État)

Un état est défini par la somme des utilités des objets rangés dans le sac-à-dos.

### Definition (État valide)

Soit un nouvel objet à ranger. Un état est dit valide s'il a un sac-à-dos de *volume libre* suffisant pour accepter ce nouvel objet.

## Definition (matrice d'états)

La *programmation dynamique* range ses résultats dans une matrice d'états $OPT$ de taille $(N \times V_{\max})$ où:

- Chaque ligne $i$ représente l'objet $o_i$,
- Chaque colonne $j$ représente le volume occupé du sac occupé,
- Chaque état de coordonnées $(i, j)$ est la somme des utilités des objets $o_k$ avec $0 \leq k \leq i$ rangés et occupant un volume $j$:

$$OPT(i,j) = \sum_{k=0}^{i} \chi_k \cdot u_k \quad \text{et} \quad \sum_{k=0}^{i} \chi_k \cdot v_k = j$$

$\chi_k \in \mathbb{B}$ indique la présence ou non de l'objet $o_k$ dans le sac.

La *programmation dynamique* range le premier objet dans tous les états *valides* de la matrice.

Elle opère de même pour le deuxième objet puis pour le troisième objet et ainsi de suite jusqu'au dernier objet.

Une fois tous les objets traités, l'état $OPT(N, V_{\text{max}})$ contiendra la solution recherchée.

## Théorème (Observations sur le dernier objet)

*Soit* $OPT(N, V_{\max})$ *représentant la solution optimale*[a]*.*

*Soit le dernier objet* $o_N = (v_N, u_N)$*. Cet objet est peut-être dans la sac-à-dos, ce qui se traduit par:*

$$\begin{cases} o_N \notin Sàd & \Longrightarrow \quad OPT_{\notin}(N, V_{\max}) = OPT(N-1, V_{\max}) \\ o_N \in Sàd & \Longrightarrow \quad OPT_{\in}(N, V_{\max}) = OPT(N-1, \boldsymbol{V_{\max} - v_N}) \boldsymbol{+ u_N} \end{cases}$$

*Il s'ensuit que:*

$$OPT(N, V_{\max}) = \max\left(\underset{\notin}{OPT}(N, V_{\max}), \mathbf{B}(v_n \leq V_{\max})\underset{\in}{OPT}(N, V_{\max})\right)$$

*où* $\mathbf{B} : \mathbb{B} \mapsto \{0, 1\}$.

[a] Exacte dans le cas du problème du paquetage ; ce qui n'est pas forcément le cas pour d'autres problèmes.

## Théorème (Par récurrence)

*Pour n'importe quel objet $o_i$ on a alors:*

$$OPT(i,V) = \max\left(OPT(i-1,V),\ \mathbf{B}(v_i \leq V)OPT(i-1,V-v_i)+u_i\right)$$

*Pour tout:*

$$\begin{cases} i & \in \{0,\dots,N\} \\ V & \in \{0,\dots,V_{\max}\} \end{cases}$$

## Remarque

*Le parcours de l'espace des solutions $OPT$ démarre avec le sac-à-dos vide ce qui se traduit par une taille de matrice de $(N+1, V_{\max}+1)$.*

| $\sharp$0 OPT | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* |
| **2** | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* |
| **3** | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* |

| $\sharp$1 OPT | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 11 | 11 | 11 | 11 |
| **2** | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* |
| **3** | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* |

| CHM | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* |
| **1** | *P* | *P* | *P* | *P* | *P* | *P* | *P* | 0 | 1 | 2 | 3 |
| **2** | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* |
| **3** | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* |

| $\sharp$2 OPT | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 11 | 11 | 11 | 11 |
| **2** | 0 | 0 | 0 | 0 | 0 | 0 | 8 | 11 | 11 | 11 | 11 |
| **3** | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* |

| CHM | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* |
| **1** | *P* | *P* | *P* | *P* | *P* | *P* | *P* | 0 | 1 | 2 | 3 |
| **2** | *P* | *P* | *P* | *P* | *P* | *P* | 0 | *P* | *P* | *P* | *P* |
| **3** | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* |

| $\sharp$3 OPT | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 11 | 11 | 11 | 11 |
| **2** | 0 | 0 | 0 | 0 | 0 | 0 | 8 | 11 | 11 | 11 | 11 |
| **3** | 0 | 0 | 0 | 0 | 5 | 5 | 8 | 11 | 11 | 11 | 13 |

| CHM | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* | *U* |
| **1** | *P* | *P* | *P* | *P* | *P* | *P* | *P* | 0 | 1 | 2 | 3 |
| **2** | *P* | *P* | *P* | *P* | *P* | *P* | 0 | *P* | *P* | *P* | *P* |
| **3** | *P* | *P* | *P* | *P* | 0 | 1 | *P* | *P* | *P* | *P* | 6 |

**Pack utility: 13 with** $\left[\textbf{Object } \sharp 3\ (4,5) \quad \textbf{Object } \sharp 2\ (6,8)\right]$

## Théorème (Complexités)

*Cet algorithme a une complexité temporelle en* $\mathcal{O}(N \times V_{\max})$.

*Toutefois, le gain obtenu par rapport à l'approche récursive se fait au détriment de l'allocation mémoire qui a une complexité en* $\mathcal{O}(N \times V_{\max})\dots$

## Rappel

`L'approche récursive a une complexité mémorielle en` $\mathcal{O}(N)$ `puisqu'elle a besoin de mémoriser` 2 `listes d'objets : le meilleur paquetage connu jusque là et le paquetage en cours de construction.`

## *Modélisation de la matrice d'états.*

```c
typedef int state_t;
struct states_array_t {
  state_t * OPT;       // Matrice d'états proprement dite
  state_t * CHM;       // Matrice des chemins dans la matrice d'états
  int num_obj, Vmax;  // Dimensions de la matrice d'états
};
// Libérer la mémoire occupée par lamatrice d'états
void free_states_array(struct states_array_t * states);
// Créer une matrice d'états ET l'initialiser à "vide"
struct states_array_t * new_states_array(const int num_objects, const int Vmax);
// Ajouter un objet dans les états "valides" de la matrice
void push_object_in_array(struct states_array_t * states,
                          struct objects_t * objects, int i);
// Visualiser la solution optimale
void view_path_array(struct states_array_t * states, struct objects_t * set);
// Visualiser la matrice d'états
void view_opt(const struct states_array_t * states);
// Visualiser les chemins associés aux états de la matrice d'états
void view_chm(const struct states_array_t * states);
```

## *Libérer et créer la matrice d'états.*

```c
void free_states_array(struct states_array_t * states) {
  assert(states);
  /** TODO **/
}

struct states_array_t * new_states_array(const int num_objects, const int Vmax) {
  struct states_array_t * NS = ...;
  assert(NS != NULL);

  /** TODO **/
  init_opt_chm(NS);
  return NS;
}
```

*Initialiser la matrice d'état.*

```c
/**
 * Private Function
 **/
void init_opt_chm(struct states_array_t * states) {
    states->OPT = ...;
    states->CHM = ...;
    for( int obj = 1; obj <= states->num_obj; obj += 1) {
        for(int bag = 0; bag <= states->Vmax; bag += 1) {
            int idx = ...;
            states->OPT[idx] = UNDTR;
            states->CHM[idx] = UNDTR;
        }
    }
    for(int bag = 0; bag <= states->Vmax; bag += 1) {
        states->CHM[bag] = UNDTR;
    }
}
```

## Ajouter un objet à la matrice d'états.

```c
void push_object_in_array(struct states_array_t * S,struct objects_t * LO,int i) {
  /* Faites attention que les objets dans LO sont rangés à partir de 0
   * tandis qu'ils sont rangés à partue de 1 dans OPT (et CHM) */
  for(int bag = 0; bag < (S->Vmax +1 ); bag += 1) {
    // Parcourir chaque état du sac-à-dos
    int pred = ...; // Calculer l'index de l'état pour l'objet (i-1)
    int curr = ...; // Calculer l'index de l'état pour l'objet (i)
    int OPT1 = S->OPT[pred];
    S->CHM[curr] = INFTY; //hyp.: l'objet i n'est pas dans le sac
    if(...) { // S'il y a de la place dans le sac
      int pred_without_i = ...; // L'index du bag SANS l'objet (i)
      int OPT2 = ...;
      if(...) { // Sélectionne la meilleur configuration
        S->OPT[curr] = ...;
        S->CHM[curr] = ...; // Noter que l'objet i est dans le sac
      } else states->OPT[curr] = ...;
    } else states->OPT[curr] = ...; // S'il n'y a pas de place
  }
}
```

## *Visualiser la solution optimale.*

```c
void view_path_array(struct states_array_t * states, struct objects_t * set) {
    int obj = states->num_obj;
    int vol = states->Vmax;
    int idx = obj * (states->Vmax + 1) + vol;
    bool nonstop = (obj == 0);
    printf("******\nTotal packaging utility : %d\n******\n", states->OPT[idx]);
    while(!stop) {
      if(states->CHM[idx] != INFTY) { // object actually put in bag
        printf("\tobjet #%d(%d, %d)\n", obj, set->objects[obj-1].volume, set->
             objects[obj-1].utility);
        stop = (states->CHM[idx] == 0);
        vol = states->CHM[idx];
      }
      obj -= 1;
      stop = stop || (obj == 0);
      idx = obj * (states->Vmax + 1) + vol;
    }
    printf("\n");
}
```

# *Visualiser la matrice d'états.*

```c
void view_opt(const struct states_array_t * states) {
  printf("OPT |\t");
  for(int bag = 0; bag < (states->Vmax + 1); bag += 1) printf("%2d\t", bag);
  printf("\n----|");
  for(int bag = 0; bag < (states->Vmax + 1); bag += 1) printf("--------");
  printf("\n");
  for(int obj = 0; obj < (states->num_obj + 1); obj += 1) {
    printf("%3d |\t", obj);
    for(int bag = 0; bag < (states->Vmax + 1); bag += 1) {
      int idx = obj * (states->Vmax + 1) + bag;
      if(states->OPT[idx] == INFTY) printf("INF\t");
      else if(states->OPT[idx] == UNDTR) printf("UND\t");
      else printf("%2d\t", states->OPT[idx]);
    }
    printf("\n");
  }
  printf("\n");
}
```

*Visualiser les chemins dans la matrice d'états.*

```
void view_chm(const struct states_array_t * states) {
  printf("CHM |\t");
  for(int bag = 0; bag < (states->Vmax + 1); bag += 1) printf("%2d\t", bag);
  printf("\n----|");
  for(int bag = 0; bag < (states->Vmax + 1); bag += 1) printf("--------");
  printf("\n");
  for(int obj = 0; obj < (states->num_obj + 1); obj += 1) {
    printf("%3d |\t", obj);
    for(int bag = 0; bag < (states->Vmax + 1); bag += 1) {
      int idx = obj * (states->Vmax + 1) + bag;
      if(states->CHM[idx] == INFTY) printf("PRE\t");
      else if(states->CHM[idx] == UNDTR) printf("UND\t");
      else printf("%2d\t", states->CHM[idx]);
    }
    printf("\n");
  }
  printf("\n");
}
```

Nous reformulons la notion d'*état* et définissons celle de *sous-espaces* pour décrire la `PD_dyn` :

**Definition (État)**

Un état est maintenant un couple $[u, v]$ où $u$ est la somme des utilités des objets *déjà* rangés dans le sac-à-dos et $v$ est le volume *occupé* dans le sac-à-dos par ces objets.

**Definition (Sous-espace)**

Le sous-espace $E_i$ contient les $n_i$ états possibles après avoir tenté (avec succès ou pas) de ranger l'objet $o_i$.

## Definition (Ajout conditionné)

Soient l'ensemble d'états $E$ et l'état $[u_i, v_i]$.

La fonction d'*ajout conditionné* suivante:

$$E \oplus \{[u_i, v_i]\} = \begin{cases} \left(E \setminus \{[u_i, v_k]\}\right) \biguplus \{[u_i, \min(v_k, v_i)]\} & \textbf{si } \exists v_k : [u_i, v_k] \in E \\ E \biguplus \{[u_i, v_i]\} & \textbf{sinon} \end{cases}$$

assure que l'ensemble des états $E$ construit contient au plus un seul état ayant une utilité de valeur $u_i$; il ne peut y avoir deux états $[u_i, v_j]$ et $[u_i, v_k]$ dans l'ensemble $E$.

De cette manière, tous les états de $E$ représentent des sommes d'utilités différentes pour un volume occupé minimal.

On a alors:

$$E_0 = \{[0,0]\}$$

Et pour tout objet $o_i = [u_i, v_i]$ tel que $1 \leq i \leq |\mathcal{O}|$ et pour tout état $[u_j, v_j] \in E_{i-1}$ ona:

$$E_i = E_{i-1} \oplus \{[u_j + u_i, v_j + v_i]\} \quad \textbf{si } v_j + v_i \leq V_{\max} \quad \forall j \in \{1, \ldots, |E_{i-1}|\}$$

Soit l'ensemble des objets initialement disponibles:

$$\mathcal{O} = \{o_1, \ldots, o_i = [u_i, v_i], \ldots, o_N\}$$

L'algorithme de la PD version *dynamique.* s'écrit alors:

$E_0 = \{[0, 0]\}$ *// Initialiser le sac à vide*

**pour** $i = 1$ **jusqu'à** $|\mathcal{O}|$

  $E_i = E_{i-1}$ *// Initialiser le* ***sous-espace*** *N°i*

  **pour tout** $[u_j, v_j]$**de** $E_{i-1}$

    **si** $v_j + v_i \leq V_{\max}$ **alors**

      $E_i = E_i \oplus [u_j + u_i, v_j + v_i]$

    **fsi**

  **fpour**

  **supprimer** $E_{i-1}$

**fpour**