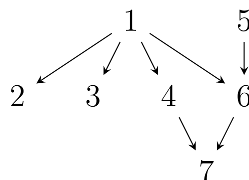
Devoir maison n°3 - À rendre le 8 mars 2026
Algorithme de Branch-and-bound pour le sac-à-dos

Notions abordées

- problème de maximisation Knapsack et son relâché fractionnaire
- résolution exacte par enumération de tous les sous-ensembles
- résolution exacte par programmation dynamique en temps pseudo-polynomial
- obtention d'un minorant grâce à un algorithme glouton
- obtention d'un majorant en resolvant le relâché fractionnaire
- résolution exacte par Branch-and-bound

Exercice 1 : Autour du problème du sac à dos.

Organisation du devoir. Dans ce devoir maison, on s'intéresse au problème d'optimisation du sac-à-dos (*knapsack* en anglais), rappelé ci-dessous. On étudiera (et implémentera en OCaml) plusieurs algorithmes pour résoudre ce problème : un algorithme d'énumération exhaustive (section 2), un algorithme de programmation dynamique (section 3), un algorithme de Branch-and-Bound (section 7). Les dépendances entre les sections de ce devoir sont résumées par le diagramme ci-dessous.



$$\text{Knapsack} : \begin{cases} \textbf{Entrée :} \ n \in \mathbb{N}, (v_i)_{i \in [\![1,n]\!]} \in \mathbb{N}_\star^n, (w_i)_{i \in [\![1,n]\!]} \in \mathbb{N}_\star^n, P \in \mathbb{N} \\ \textbf{Sortie :} \ \arg\max\{\sum_{i=1}^n v_i x_i \mid \sum_{i=1}^n w_i x_i \leqslant P\} \end{cases}$$

Pour une instance (n, v, w, P), on appelle **objets** les éléments de $[\![0, n-1]\!]$, et pour $i \in [\![0, n-1]\!]$, v_i est appelée la **valeur** de l'objet i et w_i son **poids**. Enfin P est appelé la **capacité** du sac.

Afin de représenter les instances du problème, on se munit du type `sad`[♣] défini comme suit.

```
1  type sad =
2    {
3      n  : int;                (* le nombre d'objets                *)
4      wi : int array;          (* le tableau des poids, de taille n  *)
5      vi : int array;          (* le tableau des valeurs, de taille n *)
6      p  : int                 (* le poids maximal du sac à dos, >=0  *)
7    }
```

♣. sac-à-dos

Dans toute la suite du sujet, on supposera que les objets d'une instance sont toujours triés par rapports valeur/poids décroissants. Ainsi, si e est un objet de type sad, la suite $\left(\frac{\texttt{e.vi.(i)}}{\texttt{e.wi.(i)}}\right)_{i\in[\![0,\texttt{e.n}-1]\!]}$ est décroissante. Ci-dessous un exemple de tel objet.

```
let (ex: sad) =
  {
    n  = 6 ;
    wi = [|6  ; 8 ; 10; 14; 2; 5|] ;
    vi = [|13 ; 16; 19; 24; 3; 5|] ;
    p  = 20 ;
  }
```

Sous-instances et masques. En prévision de la dernière partie de ce DM où l'on implémentera un algorithme de Branch-and-Bound pour le problème KNAPSACK, on se munit d'une notion de **sous-instance** correspondant aux restrictions imposées par le fait de choisir de prendre ou non certains objets. En effet, dans le schéma de Branch-and-Bound considéré, on est amené à résoudre des instances qui se définissent à partir de l'instance initiale par les décisions qui ont déjà été prises, par exemple la décision de sélectionner l'objet 1, ou bien la décision de ne pas sélectionner l'objet 4. Ainsi, pour une instance initiale fixée, une sous-instance est définie par un ensemble de décisions portant sur les objets. Pour chaque objet, il y a alors trois cas possibles : ou bien on l'a déjà sélectionné, ou bien on l'a déjà exclu, ou bien on n'a encore rien décidé le concernant. Pour une instance initiale de taille n, une sous-instance est donc définie par un **masque**, c'est-à-dire la donnée de n valeurs dans "oui, non, peut-être". On représente en machine les masques grâce au type suivant.

```
type masque = bool option array
```

Pour `m` un objet de type `masque` de taille n et pour $i\in[\![0,n-1]\!]$:

- si `m.`(i)= `None`, alors il n'y a aucune contrainte sur l'objet i, il peut être choisi dans la solution comme il peut ne pas l'être ;
- si `m.`(i)= `Some(true)`, alors l'objet i doit nécessairement être dans la solution ;
- si `m.`(i)= `Some(false)`, alors l'objet i doit ne peut pas être dans la solution.

Ainsi le masque `[|None; Some(true); Some(true); None; Some(false); None|]` représente une sous-instance où les objets 1 et 2 ont déjà été sélectionnés tandis que l'objet 4 a déjà été exclu.

Représentation des solutions. Une solution d'une instance de KNAPSACK est un sous-ensemble des objets de cette instance. Une telle solution peut être représentée par l'indicatrice de cet ensemble, elle même encodée par un tableau de booléens. On représente donc les solutions en machine grâce au type suivant.

```
type solution = bool array
```

Par exemple, pour la solution $\{1,2,4\}$: `[|false; true; true; false; true; false|]`.

`compagnon.ml`. On trouvera dans `compagnon.ml` : les définitions des types mentionnés dans l'énoncé, ainsi que des fonctions d'affichage pour les éléments de ces types.

1. Prise en main

Q. 1 Définir une fonction `est_sad_valide : sad -> bool` prenant en argument une instance et testant si elle est valide, en particulier si les objets sont bien classés par rapports décroissants. On ne passera pas dans le monde des flottants pour comparer deux rapports $\frac{a}{b}$ et $\frac{c}{d}$.

Solution

```ocaml
let est_sac_valide (sac : sad): bool =
  let i = ref 0 in
  while (!i < sac.n-1 && sac.vi.(!i) > 0 && sac.wi.(!i) > 0 &&
          sac.vi.(!i)*sac.wi.(!i+1) >= sac.vi.(!i+1)*sac.wi.(!i)) do
    incr i
  done;
  !i = sac.n-1 &&  sac.vi.(sac.n-1) > 0 && sac.wi.(sac.n-1) > 0 &&
  sac.n = Array.length sac.vi &&
  sac.n = Array.length sac.wi
```

Q. 2 Définir une fonction `valeur_sol : sad -> solution -> int` et une fonction `poids_sol : sad -> solution -> int` prenant en arguments une instance et une solution et retournant respectivement la valeur et le poids de cette solution.

Solution

```ocaml
let valeur_sol (sac : sad) (sol : solution) : int =
  assert(sac.n = Array.length sol);
  let res = ref 0 in
  for i=0 to sac.n-1 do
    if sol.(i) then res  := !res + sac.vi.(i) else()
  done; !res
```

```ocaml
let poids_sol (sac : sad) (sol : solution) : int =
  assert(sac.n = Array.length sol);
  let res = ref 0 in
  for i=0 to sac.n-1 do
    if sol.(i) then res  := !res + sac.wi.(i) else()
  done; !res
```

Q. 3 Définir une fonction `est_masque_valide : sad -> masque -> bool` prenant en arguments une instance et un masque et testant si le masque est valide, au sens où il représente un choix d'objets compatible avec la capacité du sac. Autrement dit, il s'agit de tester si la sous-instance correspondante admet des solutions.

Solution

```ocaml
let est_masque_valide (sac : sad) (m : masque) : bool =
  assert(sac.n = Array.length m);
  let res = ref 0 in
  for i=0 to sac.n-1 do
    match m.(i) with
    | Some(true) -> res  := !res + sac.wi.(i)
    | _ -> ()
  done;
  !res <= sac.p
```

2. Par force brute

Dans cet exercice on se propose de résoudre le problème du sac-à-dos de manière exacte par énumération exhaustive de l'espace des solutions. En vue de résoudre aussi des sous-instances avec cet algorithme, on énumère en fait, pour une instance de taille n et un masque donnés, les n-uplets de booléens qui sont compatibles avec le masque, et on cherche parmi eux celui représentant la meilleure solution.

Q. 4 Définir une fonction `next : masque -> solution -> unit` prenant en arguments un masque, un tableau de booléens et modifiant le tableau de booléens pour qu'il contienne le prochain tableau de booléens pour un ordre d'énumération bien choisi. *On pourra par exemple choisir l'ordre induit par l'énumération en binaire.* Les appels successifs à la fonction `next` avec le masque `[|Some(true); None; None; Some(false); None; None|]` peuvent conduire à la séquence de tableaux suivante.

- d'abord `[|true; false; false; false; false; false|]`
- puis `[|true; true; false; false; false; false|]`,
- puis `[|true; false; true; false; false; false|]`,
- puis `[|true; true; true; false; false; false|]`,
- puis `[|true; false; false; false; true; false|]`,
- puis `[|true; true; false; false; true; false|]`,
- puis `[|true; false; true; false; true; false|]` ...

Solution

```
exception EnumerationTerminee
let next (m : masque) (sol : solution) : unit =
  let n = Array.length m in assert (n = Array.length sol);
  let i = ref 0 in
  let fini = ref false in
  while ( not !fini && !i < n ) do
    match m.(!i)  with
    | None -> if sol.(!i) then (sol.(!i)<-false; incr i) else fini:=true
    | _ ->  incr i
```

Afin d'initialiser les énumérations on se munit aussi de la fonction suivante qui crée la plus petite solution au sens de l'inclusion qui est compatible avec le masque qu'elle prend en argument.

```
let sol_selon_masque (m : masque) : solution =
  let n = Array.length m in
  let res = Array.make n false in
  for i = 0 to n-1 do
    if m.(i) = Some (true) then res.(i) <- true
  done; res
```

On peut alors faire le jeu de tests suivant, qui correspond à l'exemple ci-dessus.

```
let test_next_exemple : unit =
  let m = [|Some(true); None; None; Some(false); None; None|] in
  let sol = sol_selon_masque m in
  assert(sol = [|true; false; false; false; false; false|]); next m sol;
  assert(sol = [|true; true; false; false; false; false|]); next m sol;
  assert(sol = [|true; false; true; false; false; false|]); next m sol;
```

```
7   assert(sol = [|true; true; true; false; false; false|]); next m sol;
8   assert(sol = [|true; false; false; false; true; false|]); next m sol;
9   assert(sol = [|true; true; false; false; true; false|]); next m sol;
10  assert(sol = [|true; false; true; false; true; false|])
```

Remarquant que pour un masque ayant k valeurs `None` on doit pouvoir énumérer 2^k solutions, on vérifie que le 2^k-ème appel à `next` lève l'exception `ExceptionTerminee` avec des tests semblables à celui qui suit.

```
1 let test_next_exception : unit =
2   let m = [|Some(true); None; None|] in
3   let sol = sol_selon_masque m in
4   for i = 1 to 3 do next done;
5   let termine_enum = try next m sol; true with
6     | EnumerationTerminee -> true
7     | _ -> false
8   in assert(termine_enum)
```

Q. 5 Définir une fonction `brute_force : sad -> masque -> (solution * int) option` calculant une solution optimale au problème Knapsack, par énumération de l'espace des solutions, pour la sous-instance représentée par les données du problème et le masque.

Solution

```
1  let brute_force (sac:sad) (m:masque) : (solution * int) option =
2    if not (est_masque_valide sac m) then None
3    else
4      let sol = sol_selon_masque m in
5      let best_sol = ref (Array.copy sol) in
6      let best_val = ref (valeur_sol sac sol) in
7      let enum_finie = ref false in
8      while (not !enum_finie) do
9        try
10         next m sol;
11         if poids_sol sac sol <= sac.p
12         then let new_val = valeur_sol sac sol in
13              if new_val > !best_val
14              then (best_sol := Array.copy sol; best_val := new_val)
15       with
16         EnumerationTerminee -> enum_finie := true
17     done;
18     Some(!best_sol,!best_val)
```

3. Par programmation dynamique

Dans cet exercice on se propose de calculer une solution optimale au problème du sac-à-dos par programmation dynamique. Dans tout cet exercice, on fixe une instance (n, v, w, P). On note alors, pour $i \in [\![0, n]\!]$ et $j \in [\![0, P]\!]$, $s_{i,j}$ la valeur optimale pour l'instance de Knapsack constituée des i premiers objets de l'instance initiale (n, v, w, P) et d'un sac de capacité j. Cette famille de valeurs

vérifie la définition récursive suivante♣.

- $\forall j \in [\![0, P]\!],\ s_{0,j} = 0$
- $\forall (i,j) \in [\![1,n]\!] \times [\![0,P]\!],\ w_i \leqslant j \Rightarrow s_{i,j} = \max(s_{i-1,j}, v_i + s_{i-1,j-w_i})$
- $\forall (i,j) \in [\![1,n]\!] \times [\![0,P]\!], w_i > j \Rightarrow s_{i,j} = s_{i-1,j}$

La valeur optimale de la solution au problème du sac-à-dos est alors $s_{n,P}$.

Q. 6 Définir une fonction `prog_dyn_tab : sad -> int array array` prenant en argument une instance et retournant le tableau des valeurs $(s_{i,j})_{i \in [\![0,n]\!], j \in [\![0,P]\!]}$ pour cette instance. On assurera une complexité pire cas en $\mathcal{O}(nP)$.

Solution

```
let prog_dyn_tab (sac: sad): int array array =
  let tab = Array.make_matrix (sac.n+1) (sac.p+1) 0 in
  for i = 0 to (sac.n) do
    for j = 0 to (sac.p) do
      if i = 0 then
        tab.(i).(j) <- 0
      else if j >= sac.wi.(i-1) then
        let best_avec_i = tab.(i-1).(j-sac.wi.(i-1)) + sac.vi.(i-1) in
        let best_sans_i = tab.(i-1).(j) in
        tab.(i).(j) <- max  best_avec_i best_sans_i
      else
        tab.(i).(j) <- tab.(i-1).(j)
    done
  done;
  tab
```

Q. 7 En déduire une fonction `prog_dyn : sad -> solution * int` prenant en argument une instance et retournant une solution optimale ainsi que sa valeur. On fabriquera cette solution à partir du tableau obtenu grâce à la fonction `prog_dyn_tab`. On assurera une complexité pire cas en $\mathcal{O}(n)$ (hors appel à `prog_dyn_tab`).

Solution

```
let sol_from_list (n:int) (l : int list) : solution =
  let sol = Array.make n false in
  List.iter (fun elem -> sol.(elem) <- true) l;
  sol

let prog_dyn (sac: sad): solution * int =
  let tab = prog_dyn_tab sac in
  let i = ref sac.n in
  let j = ref sac.p in
  let l = ref [] in
  while !i > 0 do
(*    Format.printf "%d %d@." !i !j;*)
    if tab.(!i).(!j) = tab.(!i-1).(!j) then i := !i - 1
    else begin
      l := !i :: !l ;
      j := !j - sac.wi.(!i-1) ;
```

♣. Il convient de ne pas admettre un tel résultat mais de s'assurer qu'on saurait le remontrer.

```ocaml
      i := !i - 1 ;
    end
  done;
  (sol_from_list sac.n !l), tab.(sac.n).(sac.p)
```

Organisation de la suite du sujet. En vue d'implémenter un algorithme de Branch-and-Bound pour résoudre le problème KNAPSACK, on souhaite calculer facilement des bornes sur la valeur optimale d'une instance.

- On obtient un minorant de la valeur optimale grâce à un algorithme glouton, dit **algorithme glouton entier**, qui fournit une solution valide mais pas nécessairement optimale. Cet algorithme fait l'objet de la section suivante.
- On obtient un majorant de la valeur optimale en résolvant le problème relâché $\text{KNAPSACK}_{\mathbb{Q}}$ défini ci-après.

$$\text{KNAPSACK}_{\mathbb{Q}} : \begin{cases} \textbf{Entrée :} \ n \in \mathbb{N}, \text{ deux suites } (v_i)_{i\in[\![0,n-1]\!]} \in (\mathbb{N}^\star)^n \text{ et } (w_i)_{i\in[\![0,n-1]\!]} \in (\mathbb{N}^\star)^n,\ P \in \mathbb{N} \\ \textbf{Sortie :} \ x \in ([0,1] \cap \mathbb{Q})^n \text{ tel que } \sum_{i=1}^n x_i p_i \leqslant P \text{ et maximisant } \sum_{i=1}^n x_i v_i \end{cases}$$

Les instances de ce problème sont semblables aux instances de KNAPSACK, mais l'ensemble des solutions pour une instance donnée est élargi : au lieu de devoir trancher entre prendre un objet ou ne pas le prendre, il est ici possible de prendre une fraction de chaque objet. Le poids et la valeur d'une fraction $\rho \in [0,1] \cap \mathbb{Q}$ d'un objet i sont respectivement ρw_i et ρv_i. Ce problème peut être résolu à l'optimum par un algorithme glouton dit **algorithme glouton fractionnaire**, qui fait l'objet de la section 6.

4. Algorithme glouton entier

Dans cette section, on se propose de résoudre le problème KNAPSACK de manière heuristique par une stratégie gloutonne : on prend en priorité les objets ayant le meilleur rapport valeur/poids. Cet algorithme n'est pas optimal, *i.e.* il ne fournit pas toujours une solution optimale, mais il fournit toujours une solution, et avec la valeur de cette solution un minorant de la valeur optimale.

Q. 8 Définir une fonction `glouton_n : sad -> masque -> (solution * int) option` calculant une solution (non nécessairement optimale) au problème KNAPSACK, pour la sous-instance représentée par l'instance et le masque passés en arguments. Si cette instance n'admet pas de solution le fonction devra retourner `None`. Dans le cas contraire, elle devra calculer non seulement une solution (en suivant l'heuristique gloutonne), mais aussi sa valeur. On assurera une complexité linéaire.

Solution

```ocaml
let glouton_n (s: sad) (m: masque): (solution * int) option =
  if not (est_masque_valide s m) then None
  else
    let poids_restant   = ref s.p in
    let valeur_courante = ref 0 in
    let sol = Array.make s.n false in
    (* première passe pour les objets qui *doivent* être mis selon m  *)
    for i = 0 to s.n - 1 do
      match m.(i) with
      | Some(true) ->
          sol.(i) <- true;
```

```
12          poids_restant := !poids_restant - s.wi.(i);
13          valeur_courante := !valeur_courante + s.vi.(i)
14        | _ -> ()
15      done;
16      (* seconde passe, on essaie de mettre les autres objets, en glouton*)
17      if !poids_restant <= 0 then None else
18      begin
19        for i = 0 to s.n - 1 do
20          if not (est_impose m.(i)) && s.wi.(i) <= !poids_restant
21          then begin
22            sol.(i) <- true;
23            poids_restant := !poids_restant - s.wi.(i);
24            valeur_courante := !valeur_courante + s.vi.(i)
25          end
26        done;
27      Some(sol, !valeur_courante)
28      end
```

5. Interlude : implémentation d'un module pour les rationnels

Afin d'éviter la représentation des nombres rationnels au moyen de nombres flottants, ce qui conduirait à des erreurs d'arrondis, on se propose dans cet exercice de se munir d'un module permettant la représentation et la manipulation des nombres rationnels sans passer par les flottants. Dans le fichier `compagnon.ml` on trouvera le code suivant, squelette de la définition d'un module pour les nombres rationnels.

```
1 module Q = struct
2   type t = int * int
3   (* (a, b) encode a/b*)
4   (* invariant : a ∧ b = 1 et b > 0 et si a = 0 alors b = 1 *)
5   (* NB: vu l'invariant, l'égalité syntaxique de OCaml (a, b)=(c, d)*)
6   (*     encode en fait l'égalité sémantique de Q : a/b = c/d *)
7   ...
8 end
```

Comme l'indique la définition de type ci-dessus, on veillera à maintenir la propriété invariante suivante : un élément (a, b) de type `Q.t` vérifie $a \wedge b = 1$[♣], $b > 0$ et si $a = 0$ alors $b = 1$. En contrepartie, les fonctions manipulant des objets de type `Q.t` peuvent supposer opérer sur des objets vérifiant cet invariant.

Q. 9 Compléter ce module en y ajoutant des fonctions permettant de calculer :

- l'addition de deux rationnels ;
- la différence de deux rationnels ;
- l'opposé d'un rationnel ;
- le produit de deux rationnels ;
- la division de deux rationnels ;
- le rationnel correspondant à un entier ;
- le signe (dans $-1, 0, 1$) d'un rationnel ;

et des fonctions permettant de tester :

- si un rationnel est strictement plus petit qu'un autre ;
- si un rationnel est plus petit au sens large qu'un autre ;

♣. où $\wedge$ désigne le PGCD

- si un rationnel est un entier.

Pour plus de lisibilité, on pourra définir les fonctions implémentant les opérations courantes sous forme d'opérateur infixe. Une note en fin de sujet explique comment.

Solution

```
module Q = struct
  type t = int * int
  (* (a, b) encode a/b*)
  (* invariant : a □ b = 1 et b > 0 et si a = 0 alors b = 1 *)
  (* NB: vu l'invariant, l'égalité syntaxique de OCaml (a, b)=(c, d)*)
  (*     encode en fait l'égalité sémantique de Q : a/b = c/d *)

  let rec pgcd (a: int) (b: int) =
    if b = 0 then a
    else pgcd b (a mod b)

  let normalise ((a, b): t): t =
    let val_abs_a = if a < 0 then -a else a in
    let d = pgcd (val_abs_a) b in
    (a / d, b / d)

  let ( + ) ((a, b): t) ((c, d): t) =
    normalise (a * d + b * c, b * d)

  let opp ((a, b): t): t = (-a, b)

  let ( - ) (x: t) (y: t) = x + (opp y)

  let ( * ) ((a, b): t) ((c, d): t) =
    normalise (a * c, b * d)

  let inv ((a, b): t) =
    if a = 0 then raise (Invalid_argument "inverse d'un nombre nul") else
      (b, a)

  let ( / ) (x: t) (y: t) = ( * ) x (inv y)

  let of_int (a: int) = (a, 1)
  let of_div_int (a: int) (b: int) = normalise (a, b)
  let is_int ((a, b): t) = b = 1
  let to_int ((a, b): t) =
    if b != 1 then raise (Invalid_argument "to_int d'un nombre non entier")
      else a

  let print ((a, b): t) = Printf.printf "%d/%d" a b
  let printf fmt ((a, b): t) = Format.fprintf fmt "%d/%d" a b
  let dirty_printf fmt ((a, b): t) =
    Format.fprintf fmt "%.2f" ((float_of_int a) /. (float_of_int b))
    (* retourne le signe (dans {-1, 0, 1}) de la fraction en argument *)

```

```
44   let sgn ((a, _): t) = if a < 0 then -1 else if a = 0 then 0 else 1
45
46   let ( < ) (x: t) (y: t) = sgn (x - y) < 0
47   let ( <= ) (x: t) (y: t) = (x = y) || (x < y)
48
49   let one = of_int 1
50   let zero = of_int 0
51 end
```

6. Algorithme glouton fractionnaire

Représentation des solutions fractionnaire. Pour une instance de $\text{KNAPSACK}_{\mathbb{Q}}$ de taille n, une solution est définie par la fraction sélectionnée de chaque objet, soit par la donnée de n rationnels. On représente en machine de telles solutions grâce au type suivant.

```
1 type qsolution = Q.t array
```

La stratégie gloutonne consistant à sélectionner en priorité les objets ayant le meilleur rapport valeur/poids fournit une solution optimale dans le cas du problème fractionnaire♣.

Q. 10 Définir une fonction `glouton_r : sad -> masque -> (qsolution * Q.t) option` calculant, pour une sous-instance représentée par une instance et un masque, une solution optimale au problème $\text{KNAPSACK}_{\mathbb{Q}}$. Si cette sous-instance n'admet pas de solution la fonction retournera `None`. Dans le cas contraire, la fonction calculera non seulement une solution optimale, mais aussi sa valeur. On assurera une complexité linéaire.

Solution

```
1  let glouton_q (s: sad) (m: masque): (solution_q * Q.t) option =
2    let poids_restant  = ref s.p in
3    let valeur_courante = ref 0 in
4    let sol = Array.make s.n Q.zero in
5    (* première passe pour les objets qui *doivent* être mis selon m  *)
6    for i = 0 to s.n - 1 do
7      match m.(i) with
8      | Some(true) ->
9        begin
10         sol.(i) <- Q.one;
11         poids_restant := !poids_restant - s.wi.(i);
12         valeur_courante := !valeur_courante + s.vi.(i);
13       end
14     | _ -> ()
15   done;
16   (* seconde passe, on essaie de mettre les autres objets *)
17   if !poids_restant <= 0 then None
18   else
19     begin
20       let i = ref 0 in
```

♣. Il convient de ne pas admettre un tel résultat mais de s'assurer qu'on saurait le redémontrer, par argument d'échange par exemple

```
21      let continue = ref true in
22      while (!i < s.n && !continue) do
23        if est_impose m.(!i) then i := !i + 1
24        else if s.wi.(!i) <= !poids_restant then
25          begin
26            sol.(!i) <- Q.one;
27            poids_restant := !poids_restant - s.wi.(!i);
28            valeur_courante := !valeur_courante + s.vi.(!i);
29            i := !i + 1;
30          end
31        else continue := false
32      done;
33      if !i < s.n then
34        begin
35          sol.(!i) <- Q.of_div_int !poids_restant s.wi.(!i);
36          let val_sol = Q.((of_int !valeur_courante) + (sol.(!i) * (of_int
            ↪  s.vi.(!i)))) in
37          Some(sol, val_sol)
38        end
39      else
40        Some(sol, Q.of_int !valeur_courante)
41    end
```

7. Algorithme de Branch-and-Bound

On se propose finalement d'implémenter un algorithme de type Branch-and-Bound résolvant le problème Knapsack. De la même manière que dans le cours, cet algorithme est caractérisé par les choix suivants.

- Pour chaque sous-instance, une solution réalisable (non nécessairement optimale) sera fournie par l'heuristique gloutonne, définie en **Q. 8**.
- Pour chaque sous-instance, un majorant des valeurs des solutions sera fourni par l'heuristique gloutonne du problème fractionnaire, définie en **Q. 10**.
- La stratégie de branchement sera la suivante : si une instance doit être découpée, c'est que l'optimum rationnel et l'optimum entier ne coïncident pas, ainsi l'un des objets n'est choisi que de manière fractionnaire dans l'optimum rationnel, on génère alors la sous-instance dans laquelle cet objet a été sélectionné et celle dans laquelle cet objet a été exclu. On notera `branchement` la procédure qui génère ces deux nouvelles sous-instances à partir d'une sous-instance et d'une solution fractionnaire.
- La stratégie d'exploration est de parcourir l'arbre en largeur. Ainsi la structure adaptée pour gérer les sous-instances à traiter, les nœuds à explorer, est une file. Elle pourra être implémentée grâce au module `Queue` fourni par OCaml.

L'algorithme est alors le suivant.

Algorithme 1 : Algorithme Branch-and-Bound pour le problème Knapsack

Entrée : E une instance du problème Knapsack
Sortie : Une solution optimale pour E et sa valeur

```
1  meilleure_valeur ← 0 ;
2  meilleure_solution ← ∅ ;
3  todo ← {E} ;
4  tant que todo ≠ ∅ faire
5    Soit I une instance que l'on ôte de todo ;
6    Calculer s_Q une solution du relâché fractionnaire de I et M sa valeur ;
7    si M > meilleure_valeur alors
8      Calculer s une solution de I et m sa valeur ;
9      si m > meilleure_valeur alors
10       meilleure_valeur ← m ;
11       meilleure_solution ← s ;
12     si M ⩾ m + 1 alors
13       I_1, I_2 ← branchement(I, s_Q) ;
14       Ajouter I_1 et I_2 à todo
15 retourner (meilleure_solution, meilleure_valeur)
```

Q. 11 Définir une fonction `impose_i : masque -> int -> bool -> masque` prenant en arguments un masque, un objet et un booléen, et qui retourne un **nouveau** masque[♣] obtenu en ajoutant la contrainte que l'objet doit être présent si le booléen est `true`, et qu'il ne doit pas être présent sinon.

Solution

```
let impose_i (i:int) (b:bool) (m:masque) : masque =
  let res = Array.copy m in
  res.(i) <- Some b;
  res
```

Q. 12 Définir une fonction `find_i_frac : qsolution -> int` prenant en argument une solution fractionnaire et retournant le premier objet pour laquelle la fraction indiquée n'est pas entière.

Solution

On propose deux solutions ici pour cette fonction.

```
let findi (a: 'a array) (f: 'a -> bool) : int =
  let exception Found of int in
  try Array.iteri (fun i a -> if f a then raise (Found i)) a; raise
   ↪ Not_found
  with Found i -> i

let find_i_frac (sol_q : solution_q) : int =
  findi sol_q (fun q -> not (Q.is_int q))
```

```
let find_i_frac_bis (sol_q : solution_q) : int =
  let n = Array.length sol_q in
```

♣. il est donc nécessaire de copier le masque pris en argument

```
3   let i =ref 0 in
4   while (!i < n && Q.is_int (sol_q.(!i)) ) do
5     incr i
6   done;
7   assert (!i<n);
8   !i
```

Q. 13 Définir une fonction `branch_and_bound : sad -> solution * int` implémentant l'algorithme précédent.

Solution

```
1  let branch_and_bound (sac: sad): solution * int =
2    let todo = Queue.create () in
3    let masque_initial = Array.make sac.n None in
4    Queue.push masque_initial todo;
5    let best_val = ref 0 in
6    let best_sol = ref (Array.make sac.n false) in
7    while not (Queue.is_empty todo) do
8      Queue.iter affiche_masque todo;
9      Unix.sleepf 0.4;
10     let mask = Queue.pop todo in
11     match glouton_n sac mask, glouton_q sac mask with
12     | None, _ ->
13       Format.printf "PAS DE SOLUTION@.";
14       Format.printf "====================@."
15     | _, None -> assert false
16     | Some (sol_n, value_n), Some(sol_q, value_q) ->
17       Format.printf "%d %a@." value_n Q.dirty_printf value_q;
18       Format.printf "====================@.";
19       if value_n > !best_val then
20         begin
21           best_val := value_n;
22           best_sol := sol_n;
23         end;
24       if Q.(value_q < (of_int !best_val)) then ()
25         (* on a déjà trouvé mieux que ce que value_q laisse espérer*)
26       else if Q.(value_q < (of_int (succ value_n))) then ()
27         (* value_q ne laisse pas espérer mieux localement que value_n*)
28       else
29         begin
30           let i_frac = findi sol_q (fun q -> not (Q.is_int q)) in
31           Queue.push (impose_i (i_frac) true mask) todo;
32           Queue.push (impose_i (i_frac) false mask) todo;
33         end
34   done;
35   Format.printf "Solution optimale de valeur %d :\n" !best_val;
36   affiche_sol !best_sol;
37   !best_sol, !best_val
```

Opérateurs binaires à notation infixe en OCaml

En OCaml, les opérateurs arithmétiques usuels comme `+`,`*`,`/`,`mod` ...ou `+.`,`*.`,`/.` ..., les opérateurs booléens `&&` et `||` ou encore les opérateurs de concaténation[a] `@` et `^` sont en fait des fonctions de deux arguments qui ont la particularité de pouvoir être appelées selon la notation infixe, c'est-à-dire en étant placé entre leur deux arguments et non à gauche de ceux-ci. Afin de désigner de telles fonctions en OCaml, on encadre l'opérateur de parenthèses. Par exemple `(+)` est une expression de type fonctionnel `int -> int -> int`, alors que l'expression `+` n'est pas syntaxiquement correcte. De même :

- `(mod)` est de type `int -> int -> int`;
- `(+.)` est de type `float -> float -> float`;
- `(&&)` est de type `bool -> bool -> bool`;
- `(^)` est de type `string -> string -> string`;
- `( |> )` est de type `'a -> ('a -> 'b) -> 'b`;
- ...[b]

De tels opérateurs ne sont pas réservés à la librairie standard, on peut définir de tels opérateurs à condition de choisir un nom de fonction valide. La description précise des noms valides est donnée dans la documentation : `https://v2.ocaml.org/manual/expr.html`, rubrique *Operators*. Sans être exhaustif, avec $\Sigma_1 = \{\$, \&, *, +, -, /, =, >, \hat{}, |, \%, <\}$ et $\Sigma_2 = \Sigma_1 \cup \{\sim, !, ?, :, .\}$, les noms dans $\Sigma_1 \cdot \Sigma_2^*$ sont valides.

Exemple. On pourrait par exemple définir l'opérateur `%%` pour le produit scalaire sur les vecteurs d'entiers encodés par des tableaux comme suit.

```
1  exception ProduitScalaireImpossible
2  let ( %% ) (x: int array) (y: int array) : int =
3    if Array.length x = Array.length y
4    then
5      let res = ref 0 in
6      for i = 0 to Array.length x - 1 do
7        res:= !res + x.(i) * y.(i)
8      done; !res
9    else raise ProduitScalaireImpossible
```

Après une telle définition le jeu de tests suivant est vérifié.

```
1    assert ([|0; 0; 0|]    %% [|1; 2; 3|]    = 0 );
2    assert ([|0; 1; 0|]    %% [|1; 2; 3|]    = 2 );
3    assert ([|1; 1; 1; 1|] %% [|1; 2; 3; 4|] = 10)
```

a. Attention le symbole `::` n'est pas un opérateur de concaténation, en effet c'est un constructeur du type `'a list` (même s'il a un statut particulier qui lui permet d'être placé entre ses paramètres).

b. Les seules exceptions sont `( * )` et `( *. )` qu'il faut écrire en laissant un espace entre la parenthèse et l'étoile car l'enchaînement `(*` est réservé pour marquer un début de commentaire.