

Benjamin Kataliko Viranga
8842942
CSI2520

Partie IV - Projet Intégrateur - Scheme

→ **Fichiers** : bruteForce.rkt , p1.txt

→ **Output** :

```
bruteForce.rkt ▾ (define ...) ▾  
  
#lang scheme  
  
; Student : Benjamin Kataliko Viranga  
; Student ID : 8842942  
; CSI2520  
; Projet Intégrateur - Partie Fonctionnelle (Scheme)  
  
Welcome to DrRacket, version 8.0 [cs].  
Language: scheme, with debugging; memory limit: 128 MB.  
> (solveKnapsack "p1.txt")  
  
> Collected capacity : 7  
> Collected items : ((A 1 1) (B 6 2) (C 10 3) (D 15 5))  
  
> Optimal subset found : ((B 6 2) (D 15 5))  
> Optimal solution : (21 (B D))  
  
(21 ("B" "D"))  
>
```

Fonctions utilisées

- **solveKnapsack filename** : fonction principale prenant en paramètre le nom du fichier contenant les données du Knapsack afin de trouver la solution optimale correspondante.

```
; solveKnapsack  
(define (solveKnapsack filename)  
  (let(  
    (src_dest (string-append filename ".sol"))      ; .sol file pour la  
    solution final  
    (optimal_sol (process-content (get-filtered-content filename)) )  
    ; collecte la solution optimal  
  )  
    (begin
```

```

        (write-to-a-file src_dest list (car optimal_sol) '(\n) (cdr
optimal_sol)) ; écrire la solution optimale
        optimal_sol
    )))

```

- **write-to-a-file** *dest text* : permet d'écrire dans le fichier *dest* le texte *text*.

```

(define (write-to-a-file path txt)
  (call-with-output-file path
    (lambda (output-port)
      (write txt output-port))
    #:exists 'replace ))

```

- **get-filtered-content** *filename* : s'assure que la liste des données obtenues de la fonction **get-content** ne contienne pas de caractère **#\return** **#\space** (ou espace) **#\newline**. Pour seulement ces valeurs ? Parce que dans le **get-content** en faisant un **read-char** au lieu d'un **read**, ces caractères apparaissent.

```

; list without #\space #\newline #\return

(define (get-filtered-content filename)
  (filter-symbol (remv* ' (#\return #\space #\newline) (get-content
filename)))
)

```

- **filter-symbol** *L* : Lors de la lecture du fichier avec le **get-content** , les noms des items sont introduits dans la liste de retour en tant que symbole avec la fonction **read**. Cette fonction **filter-symbol**, permet de transformer tout symbole dans la liste collectée en string.

```

; change all symbols in a list to string
(define (filter-symbol L)
  (cond
    [(null? L) '()]
    [(symbol? (car L)) (cons (symbol->string(car L)) (filter-symbol (cdr
L))) ]
    [else (cons (car L) (filter-symbol (cdr L))) ]
  )
)

```

- **get-content** *filename* : parcourt le fichier d'entrée et retourne son contenu dans une liste.

```
; Prendre le contenu du fichier
(define (get-content filename)
  (call-with-input-file filename
    (lambda (input-port)
      (let loop ((x (read input-port)))
        (if (not (eof-object? x))
            (begin
              ; (display x)
              (cons x (loop (read input-port)))
            )
            empty
          )
      )
    )
  )
)
```

- **process-content**: collecte les valeurs des capacités et des items, et appelle la fonction **knapsack** pour procéder avec la solution optimale. *get-last* retourne le dernier élément de la liste **content**. Initialement, la liste **content** contient à son premier index (index 0), le nombre d'éléments à ajouter dans le knapsack et à son dernier index la capacité du knapsack. Ce qui reste entre le premier et le dernier item concerne les items à ajouter dans le knapsack, soit (*remove-first (remove-last content)*).

```
; process-content
; content is a list of the data collected from the file
(define (process-content content)
  (
    let (
      (capacity (get-last content))
      ; (unprocessed_items (remove-first (remove-last content)))
      (processed_items (process-items (remove-first (remove-last
content)) ) )
    )
    (begin
      (display "\n")
      (display "> Collected capacity : ")
      (display capacity)
      (display "\n")
      ; (display unprocessed_items)
      (display "> Collected items : ")
      (display processed_items)
    )
  )
)
```

```

    (display "\n")
    (knapsack capacity processed_items)
  )
)
)

```

- **process-items** : retourne les listes d'items collectés groupés en sous-groupe de trois sous la forme ((name, value, weight),...).

```

; process items
; return a list of items in the items = ((name, value, weight),...)
; assumant que la taille de la liste items est un multiple 3
(define (process-items items)
  (if( = (length (cdr(cdr(cdr items)))) ) 0)
    (cons (list (car items) (car (cdr items)) (car (cdr (cdr items))))
empty)
    (cons(
      list (car items) (car (cdr items)) (car (cdr (cdr items)))
      ) (process-items (cdr(cdr(cdr items)))) )
    )))

```

- **subsets** : permet de trouver toutes les combinaisons possibles des items d'une liste.

```

; subset of given list
; reference : https://gist.github.com/vishesh/5731608
(define (subsets s)
  (if (null? s)
    (list null)
    (let ((rest (subsets (cdr s))))
      (append rest (map (lambda (x)
                          (cons (car s) x))
                        rest)))))

```

- **process-subsets** *subs*: permet d'obtenir la liste de tous les poids des subsets de knapsack obtenus à partir de la liste *subs*.

```

; process de subsets of items and return the list of the optimal weight
calculated for
; each subset
(define (process-subsets subs)
  (cond
    [(null? (car subs)) (cons -1 (process-subsets (cdr subs)) )])
    [(null? (cdr subs)) (cons (process-sublist (car subs)) empty)]
  )

```

```

    [ else (cons (process-sublist (car subs)) (process-subsets (cdr
subs))) ]
  )
)

```

- **process-sublist sList** : permet de calculer les poids des items de sList dans un knapsack sans tenir compte de la capacité maximale du sac.

```

; process the list inside the list of subsets
; assuming sList is not null
(define (process-sublist sList)
  ( if(null? (cdr sList))
    (get-last (car sList))
    (+ (get-last (car sList)) (process-sublist (cdr sList) ))
  )
)

```

- **sums_values** : permet d'obtenir la somme des valeurs d'un set d'item pour le knapsack.

```

; get the names of items inside the list of items
; sum the values inside the list of items
; the subset always have 3 items
; item name, item value, item weight
; the index of the item value is therefore 1
; assuming the optimal_set is never null
(define (sums_values optimal_set)
  (cond
    [(null? (cdr optimal_set)) (list-ref (car optimal_set) 1)]
    [else (+ (list-ref (car optimal_set) 1) (sums_values (cdr
optimal_set)) ) ]
  )
)

```

- **get_items_names** : permet d'obtenir les noms des items à partir d'un set d'items destinés au Knapsack.

```

; get items names in the optimal subset
; assuming the optimal_set is never null
(define (get_items_names optimal_set)
  (cond
    [(null? (cdr optimal_set)) (cons (list-ref (car optimal_set) 0)
empty)] ; item name is always the first element of the subset

```

```

    [else (cons (list-ref (car optimal_set) 0) (get_items_names (cdr
optimal_set)) ) ]
  )
)

```

- **process_optimal_solution** : permet de construire la liste de la solution finale avec la valeur optimale et les noms des items.

```

; process optimal solution
; put the optimal value at the beginning of the list
; and the items names as a sublist
(define (process_optimal_solution optimal_set)
  (cons (sums_values optimal_set) (cons (get_items_names optimal_set)
empty) )
)

```

- **get-legal-knapsack wList max_cap** : permet de collecter les valeurs de poids "légaux" de knapsack c'est-à-dire les poids contenus dans *wList* qui sont inférieurs ou égaux à *max_cap*.
wList (weights list) est obtenue via la fonction **process-subsets**.

```

; keep the weights that are lower to the maximum capacity
(define (get-legal-knapsack wList max_cap)
  (begin
    ; (display "\n")
    ; (display wList) ; debug purpose
    ; (display "\n")
    (cond
      [(and (< (car wList) 0) (not(null? (cdr wList)))) (cons 0
(get-legal-knapsack (cdr wList) max_cap))]
      [(and (< (car wList) 0) (null? (cdr wList))) (cons 0 empty)]
      [( and (>= (car wList) 0) (<= (car wList) max_cap) (null? (cdr
wList)) ) (cons (car wList) empty)]
      [( and (>= (car wList) 0) (<= (car wList) max_cap) (not(null? (cdr
wList)))) (cons (car wList) (get-legal-knapsack (cdr wList) max_cap))]
      [( and (>= (car wList) 0) (>= (car wList) max_cap) (not(null? (cdr
wList)))) (cons 0 (get-legal-knapsack (cdr wList) max_cap))]
      [ else (cons 0 empty) ]
    )
  ))

```

- **knapsack** : trouve la solution optimale pour le knapsack de capacité *capacity* avec la liste *items*.

```
; knapsack
(define (knapsack capacity items)
  (let (
    ; get the list of the possible subsets
    (all_subsets (subsets items))
    ; get the list of legal weights among all the possible solution
    (legal_weights (get-legal-knapsack (process-subsets (subsets
items) ) capacity))
  )
  (begin
    ;(display "\n")
    (let(
      ; get index of the maximum legal weight found in the
legal_weights list
      (max_index (get-list-index legal_weights (maximum
legal_weights)))
    )
      (begin
        ;;(display "\n")
        ; (display max_index)
        (display "\n")
        (let (
          ; get the index of the optimal subset
          (optimal_subset (list-ref all_subsets max_index))
        )
          (display "> Optimal subset found :")
          (display optimal_subset)
          (display "\n")
          (display "> Optimal solution : ")
          (display (process_optimal_solution optimal_subset))
          (display "\n")
          (display "\n")
          (process_optimal_solution optimal_subset)
        ))))))
```

Note : Les références spécifiques sont dans le fichier **bruteForce.rkt** au-dessus des fonctions concernées.

General reference :

- <https://docs.racket-lang.org/>