Projet individuel
d'algorithmique-programmation

IPF : groupe 3 (Enseignant : C. Mouilleron)

Étude de différentes approches pour résoudre le probleme : SUBSET-SUM-OPT

Prénom : Guangyue

Nom : CHEN

Table des matières

1 L’introduction

(1) Préambule

Ce projet est à réaliser en OCaml individuellement . L'objectif est de étudier des différentes approches pour résoudre le problème du SUBSET- SUM-OPT .

(2) Sujet

le problème de SUBSET- SUM-OPT

Étant donnés un ensemble fini E d’entiers strictement positifs et un entier cible s , trouver l’entier $s' \leq s$ le plus grand possible tel qu’il existe un sous-ensemble $E' \subseteq E$ vérifiant $\sum_{e \in E'} e = s'$.

C’est à dire qu’il faut retour une somme d’un sous-ensembles de E , l’objectif de ce projet est de tester quatre approches possibles pour résoudre ce problème . Ce rapport présente différentes approches pour répondre au problème, proposé par le sujet. Après l'étude d’approche, ils seront comparées. Les approches:

APPROCHE NAÏVE

APPROCHE PLUS DIRECTE

APPROCHE AVEC NETTOYAGE

APPROCHE DE TYPE DIVISER POUR RÉGNER

Dans la 3ème partie , je vais les présenter plus clairement et montrer leur réalisation .

2 Les résultats

(1) État du travail

16/03/18 -> 20/03/18 : 1ère implémentation du code

Complète les codes de la question 1 à la question 12 , J'ai réalisé ces quatre méthodes et les ai testé .

24/03/18 -> 26/03/18 : 2ère implémentation du code

Récrire les fonctions précédents pour réaliser la question 13 , (les fonctions peuvent rentre la somme d'un sous-ensemble et une liste qui contenant les entiers de ce sous-ensemble) . Et après je compare les quatre approches et écris les interfaces.

31/03/18 -> 01/04/18 : Écrire du rapport

(2) Les résultats et les limites

Après deux semaines de travail, j'ai réussi à réaliser quatre méthodes différentes par deux formats différents qui retourne un INT ou un COUPLE . Après les tests des quatre méthodes , je trouve qu'elles ont bien fonctionné et ont retourné la même valeur correcte. Le deuxième format peut retourner des différents sous-ensembles, mais la somme est égale.

Cependant , il subsiste des limites : quand je teste les codes, je trouve que quand les sous-ensembles sont trop grands , pour l'approche naïve et l'approche plus directe , On a un *"Stack overflow during evaluation*" . Mais l'approche avec nettoyage marche toujours ne présente aucun problème ou message d'erreur. Je suppose que c'est parce que je fais trop d'appels d'une fonctions récursive et la pile d'exécution a été dépassé .

3 La réalisation.

(1) Approche naïve : subset_sum_0

Cette approche consiste à déterminer tous les sous-ensembles E' de E, d'effectuer la somme sur tous les E', et de renvoyer la somme la plus proche de s .

Question 1 : somme sur un ensemble

Pour cette question, j'ai utilisé la fonction fold_right ,

```
let sum l=List.fold_right ( function x-> function acc -> x+acc) l 0 ;;
```

Donc la complexité est en O(n) . Le test avec une liste [1; 2; 3; 4; 5; 6; 7; 8] retourne int = 36.

Question 2 : génération des parties d'un ensemble

Ici je crée trois fonctions pour générer des parties d'un ensemble : merge , powerset1 et powerset .

La fonction merge prend deux paramètres , une liste de sous-ensemble (une liste de liste) et un entier , il va ajouter cet entier dans toutes les listes qui sont dans la liste l .

```
let rec merge x l = match l with
l[]->[]
la::r->(x::a)::(merge x r);;
```

La fonction powerset1 appelle la fonction merge , il calcule les sous-ensemble des parties de E . Soit $x \in E$, $P \leftarrow$ Sous–ensembles ($E \setminus \{x\}$) , il retourne $P \cup \{E' \cup x \mid E' \in P\} \cup \{x\}$.

```
let rec powerset1 l = match l with
l[]->[]
la::r->([a]::(powerset1 r))@(merge a (powerset1 r))
    ;;
```

La fonction powerset ajoute un vide ensemble dans le résultat de la fonction powerset1 .

```
let powerset l= let ml =powerset1 l in ( [ ] )::ml
;;
```

Question 3 :résolution de SUBSET_SUM_OPT par force brut

la fonction subset_sum_0 , l'idée est : Soit P ← Sous – ensembles(E) ,retourne le maximum somme(E') , ou E' ∈ P .

Ici je utilise la fonction fold_right aussi .

```
let rec  subset_sum_1 l s=List.fold_right
(function x -> function y -> if x>y&&x<=s then x else y)
(get_all_sums l) 0;;
```

(2) Approche plus directe: subset_sum_1

Dans cet approche, au lieu de calculer tous les sous-ensembles, puis de regarder la somme des éléments pour chacun d'entre eux, on se propose de calculer directement l'ensemble des sommes atteignables.

Question 4 : sommes atteignables

La fonction get_all_sums calcule les sommes , donc il retourne une liste d'entiers . L'idée est : Si un ensemble E est vide retourne vide , sinon , soit x ∈ E , S ← get_all_sums(E \ {x}) , retourne S ∪ {x + s | s ∈ S} .

```
let rec get_all_sums l =
  match l with
  l[]->[0]
  la::r ->(get_all_sums r)@(  List.map(function x->x+a) (get_all_sums
r)  );;
```

Question 5 : résolution de SUBSET_SUM_OPT

Cette fonction est simple, elle trouve la plus grande valeur dans une liste mais avec la limite L'idée est : Soit S ← get_all_sums(E) , retourne max { $s' \in S \mid 0 \leq s' \leq s$ } ;

```
let rec  subset_sum_1 l s=List.fold_right
(function x -> function y -> if x>y&&x<=s then x else y)
(get_all_sums l) 0;;
```

(3) Approche avec nettoyage: subset_sum_2

Cette approche est une variante de l'approche précédente consiste à appliquer un nettoyage au fur et à mesure du calcul des sommes atteignables.

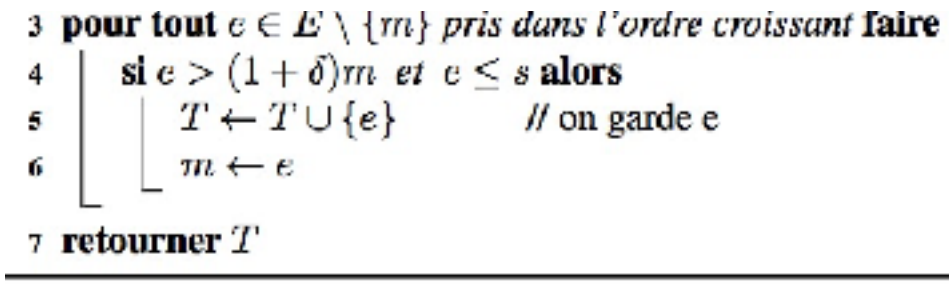
Question 6 ： fonction clean_up

Pour résoudre cette problème , je crée une fonction pour trier la liste , ici je utilise **Tri par insertion :** les fonctions insert et sort . L'idée de la fonction clean_up est : Au début, je trie la liste que je veux nettoyer . Donc la plus petite valeur est toujours la première . Et l' algorithme est déjà donné dans le sujet.

Algorithme 3 : nettoyage

Entrée : un ensemble E non vide, un entier s et une précision $\delta \in \mathbb{R}_+^*$
Sortie : un ensemble $E' \subseteq E$

1 $m \leftarrow$ plus petit élément de E
2 $T \leftarrow \{m\}$



```
3 pour tout e ∈ L \ {m} pris dans l'ordre croissant faire
4     si e > (1 + δ)m et e ≤ s alors
5         T ← T ∪ {e}          // on garde e
6         m ← e
7 retourner T
```

Voici code en OCaml :

```
let clean_up l s delta = let ml = sort l in match ml with
l [] ->[]
la::r -> List.fold_left
   (function liste_t -> function value -> let m = match (List.rev liste_t) with
[]-> s llast::r->last in
      if (float_of_int value)>(1.0+.delta)*.(float_of_int m)&&value<=s then
liste_t@[value] else liste_t)
   [a] r
;;
```

Testez la fonction avec s = 90, δ = 0.1 et la liste des entiers de 1 à 100 :

Pour tester cette fonction , je crée une fonction cree pour créer la liste des entiers de 1 à 100 .

(1) Le premier Itératif , ici je utilise la fonction fold_left , la liste que on veut est que [1] (marque List_res)

(2) le 2ème itératif , la fin de List_res est 1 , 2 > (1 + 0.1) * 1 , ajoute 2 dans List_res , [1 ; 2] .

(3) le 3ème itératif , la fin de List_res est 2 , 3 > (1 + 0.1) * 2 , ajoute 2 dans List_res , [1 ; 2 ; 3]

.
.
.

(11) le 11ème itératif , la fin de List_res est 10 , 11 = (1+0.1) *10 , n'ajoute pas 11 .
(12) le 11ème itératif , la fin de List_res est 10 , 12 > (1+0.1) *10 , ajoute 2 dans List_res [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 12]

.
.
.

(fin)[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 12; 14; 16; 18; 20; 23; 26; 29; 32; 36; 40; 45; 50; 56; 62; 69; 76; 84]

Question 7 : résolution de SUBSET_SUM_OPT

L'idée est : Si un ensemble E est vide retourne 0 . Sinon , soit x ∈ E
S ← get_all_sums_2(E \ { x }) , retourne clean_up(S ∪ {x + s I s ∈ S}) .

```
let rec subset_sum_2x l s delta=
match l with
  l[]->[0]
  la::r ->  let xl = subset_sum_2x r s delta  in
  clean_up (xl@(  List.map(function x->x+a) xl ) ) s delta
;;
```

Quand je la teste avec différent δ , je trouve que quand δ est plus petit , le résultat est différent par rapport à l'approche plus directe . Donc on remarque que le choix du paramètre δ est important pour cette approche , parce que quand δ est trop grand , le résultat a un bias , mais si δ est trop petit , la complicité est élevée . Par exemple:

Teste s = 90, δ = 0.01 et la liste des entiers de 1 à 100 - : int = 90
Teste s = 90, δ = 0.1 et la liste des entiers de 1 à 100 - : int = 84

(4)Approche de type Diviser pour régner: subset_sum_3

Question 8 : is_feasible

On nous demande ici de créer une fonction qui sur la donnée d'un entier s, d'une liste l_1 d'entier croissant, d'une liste l_2 d'entier décroissant, renvoie true si s s'écrit comme la somme d'un élément de l_1 et d'un élément de l_2, et false sinon.

L'idée est : Si les deux listes sont vide , on retourne **False** , pour chaque membre **e1** de la première liste , calcule **e1 + e2** (ou e2 est l'élément de la 2ème liste) ,jusqu'à on trouve qu'il existe **e1 + e2 = s** , retourne **true** , sinon **false** .

```
let  rec is_feasible somme liste1 liste2 = match liste1 with
l[]->false
la::r-> let l = List.map (function x -> x+a ) liste2 in
          match l with
          l[] -> false
          laa::r2-> let truth =is_feasible somme r liste2 in
             if ( aa<somme ) then  truth
            else
                 let res= List.fold_right (function x -> function y -> if x!
=somme then (0+y) else (1+y) ) l 0 in
                 if res==0 then false else true
              ;;
```

Teste la fonction :

```
is_feasible 12 [1;2;4;7] [10;8;2] ;;
: bool = true
is_feasible 16 [1;2;4;7] [10;8;2];;
: bool = false
```

Question 9 : best_feasible

Au regard de la Question 8, on nous demande maintenant de créer une fonction qui sur la donnée d'un entier s, d'une liste l1 d'entier croissant, d'une liste l2 d'entier décroissant, renvoie le plus grand entier $s' \leq s$ somme d'un élément de l1 et d'un élément de l2 .

L'idée est : Si les deux listes sont vide , on retourne **0** , pour chaque membre **e1** de la première liste , calcule **s' = e1 + e2** (ou e2 est l'élément de la 2ème liste) , on choisit la valeur qui est plus proche de s .

```
let  rec best_feasible somme liste1 liste2 = match liste1 with
l[]->0
la::r-> let l = List.map (function x -> x+a ) liste2 in
          match l with
          l[] -> 0
          laa::r2-> let res  =best_feasible somme r liste2 in

                    let xx= List.fold_right (function x -> function y -> if
x<=somme then x else y ) l aa in
                    if( res<=somme&&(somme-res)<(somme-xx) ) then res
else xx ;;
```

tester somme = 16 , liste1 = [1;2;4;7] , liste2 =[10;8;2]

```
best_feasible 12 [1;2;4;7] [10;8;2] ;;
-: int = 15
```

Question 10 : résolution de SUBSET_SUM_OPT

Pour cette approche , je crée deux fonction separer et subset_sum_3 , separer sert à séparer la liste ,et elle retourne deux listes croissantes , elle rappelle la fonction sort . La réalisation est simple .

```
let rec separer l = match l with
[]->([],[])
la::b::r->let (l1,l2) =separer r in ((a::l1),(b::l2))
la::r->let (l1,l2) =separer r in ((a::l1),l2);;
```

Et pour subset_sum_3 , il utilise la fonction sort , List.rev et get_all_sums L'idée est :

(l1, l2) ← split(l)
(n1, n2) ← (get_all_sums(l1), get_all_sums(l2))
trie n1 à croissant ,n2 à décroissant .
retourne best_feasible(s, n1, n2)

```
let rec subset_sum_3 l s = let (l1,l2) =separer l in
  let n1=sort (get_all_sums l1 )in
  let n2=List.rev (sort (get_all_sums l2 )) in
  best_feasible s n1 n2 ;;
```

(5)Comparaison des approches et amélioration

Question 11 : gen_random

Cette fonction est : sur la donnée de deux entiers naturels n et m, retourne un ensemble fini à n éléments contenant des entiers entre 0 et m - 1 inclus . La réalisation est simple, elle utilise la fonction Random.int :

```
let rec gen_random n m = if n=0 then []
      else let ml = gen_random (n-1) m in
        let x= Random.int(m-1) in
         x::ml ;;
```

Question 12 tester davantage les différentes approches

```
let test_liste1=gen_random 4 10 ;;
-: val test_liste1 : int list = [1; 2; 5; 4]
let res0 = subset_sum_0 test_liste1 20;;          res0 : int = 12
let res1 = subset_sum_1 test_liste1 20;;          res1 : int = 12
let res2 = subset_sum_2 test_liste1 20 0.01;;     res2 : int = 12
let res3 = subset_sum_3 test_liste1 20;;          res3 : int = 12

let test_liste2=gen_random 10 25 ;;
-: val test_liste2 : int list = [18; 14; 22; 0; 21; 5; 11; 16; 6; 20]
let res0 = subset_sum_0 test_liste2 40;;          res0 : int = 40
let res1 = subset_sum_1 test_liste2 40;;          res1: int = 40
let res2 = subset_sum_2 test_liste2 40 0.01;;     res2: int = 40
let res3 = subset_sum_3 test_liste2 40;;          res3 : int = 40
```

les avantages et les inconvénients

APPROCHE NAÏVE

Cette méthode est plus facile à comprendre , mais la complicité est trop élevé : $O(2^{n*}n)$. Il y a beaucoup de calculs qui ne sont pas nécessaire .

APPROCHE PLUS DIRECTE

L'approche plus directe a une allure exponentielle (en $O(2^n)$), c'est encore élevé.

APPROCHE AVEC NETTOYAGE

Cette méthode est la plus optimale. Mais il faut choisir le paramètre δ par nous même.

APPROCHE DE TYPE DIVISER POUR RÉGNER

Cette méthode est optimale , mais je trouve que l'approche diviser pour régner prend plus de temps , quand je utilise les quatre approches ensemble , les autres retourne directement , mais pour subset_sum_3 je dois attendre un peu .

Conclusion : Pour les méthodes naïve, plus directe, et de type diviser pour régner, le résultat obtenu est optimal. Bien que ces algorithmes sont moins optimaux que l'approche avec nettoyage, l'optimalité du résultat est assuré.

(6) Question 13 : Modifier les codes

Ici toutes les idées sont similaires à avant , mais il faut retourner un couple de **INT * INT LIST** ,les codes sont dans l'annexe (Page 22). Ici je montre que les résultats des tests :

```
(*Question 13*)                                                    19

let res0 = subset_sum_0 test_liste1 20;;

let res1 = subset_sum_1 test_liste1 20;;

let res2 = subset_sum_2 test_liste1 20 0.01;;

let res3 = subset_sum_3 test_liste1 20;;

(* tester avec l=test_liste1 ,s =20
 res0 : int * int list = (12, [1; 2; 5; 4])
 val res1 : int * int list = (12, [1; 2; 5; 4])
val res2 : int * int list = (12, [1; 2; 5; 4])
val res3 : int * int list = (12, [1; 5; 2; 4])  *)


let test_liste2=gen_random 10 25 ;;

let res0 = subset_sum_0 test_liste2 9;;

let res1 = subset_sum_1 test_liste2 9;;

let res2 = subset_sum_2 test_liste2 9 0.01;;

let res3 = subset_sum_3 test_liste2 9;;


(*tester avec l=test_liste2 ,s =9
val res0 : int * int list = (8, [5; 0; 3])
 val res1 : int * int list = (8, [5; 0; 3])
val res2 : int * int list = (8, [5; 3])
val res3 : int * int list = (8, [3; 5; 0])*)
```

4 annexes.

(*Question 1*)

```
(** sum : int list -> int
@param l Une liste des entiers
@return la somme des elements de l
*)

(* tester list =[1; 2; 3; 4; 5; 6; 7; 8]
- : int = 36*)
```

```
 (*Question 2*)
```

```
(** merge : 'a -> 'a list list -> 'a list list
@param x un entier , l Une liste
@return une liste de 'a liste qui contient x dans chaque 'a liste*)

(** powerset1 : 'a list -> 'a list list
@param l Une liste
@return une liste de 'a liste qui contient tous les sou-ensembles du liste l ,sauf '[]'
*)
```

```
(** powerset : 'a list -> 'a list list
@param l Une liste
@return une liste de 'a liste qui contient tous les sou-ensembles du liste l , contient '[]'
*)

(* tester list =[1; 3; 4; 7]
int list list =[[]; [1]; [3]; [4]; [7]; [4; 7]; [3; 4]; [3; 7]; [3; 4; 7]; [1; 3]; [1; 4];
 [1; 7]; [1; 4; 7]; [1; 3; 4]; [1; 3; 7]; [1; 3; 4; 7]]*)
```

(*Question 3*)

```
(** subset_sum_0 : int list -> int -> int = <fun>
@param l Une liste des entiers ,s un entier
@return un entier qui est le plus grand somme des sous-ensembles de l mais inderieur à s
*)

(* tester a la fin ensemble*)
```

(*Question 4*)

```
(** get_all_sums : int list -> int list
@param l Une liste des entiers
@return une liste des entiers qui cotient tous les sommes des sous-ensembles de l *)
```

(* tester list =[1; 3; 4; 7]

int list = [0; 7; 4; 11; 3; 10; 7; 14; 1; 8; 5; 12; 4; 11; 8; 15]*)

(*Question 5*)

(** subset_sum_1 : int list -> int -> int

@param l Une liste des entiers ,s un entier

@return un entier qui est le plus grand somme des sous-ensembles de l mais inderieur à s *)

(* tester a la fin ensemble*)

(*Question 6*)

(** insert : 'a -> 'a list -> 'a list

@param x un element,l Une liste croissante

@return une liste croissante qui contient le element x

*)

(** sort : 'a list -> 'a list

@param l Une liste

@return une liste croissante avoir les meme elements

```
*)


(* testerlist =[1; 7; 4; 3]
int list = [1; 3; 4; 7]*)

(** clean_up : int list -> int -> float -> int list
@param l Une liste des entiers , s un entier ,delta un float
@return une liste apres lèalgorithme nettoyage
*)

(** cree : int -> int list
@param i un entier
@return une liste contient les entiers de i à 100
*)

(* tester s = 90, δ = 0.1 et la liste des entiers de 1 a` 100
- : int list =
[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 12; 14; 16; 18; 20; 23; 26; 29; 32; 36; 40;
 45; 50; 56; 62; 69; 76; 84]*)
clean_up (cree 1) 90 0.1;;


(* tester s = 90, δ = 0.01 et la liste des entiers de 1 a` 100
- : int list =[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19; 20; 21;
 22; 23; 24; 25; 26; 27; 28; 29; 30; 31; 32; 33; 34; 35; 36; 37; 38; 39; 40;
 41; 42; 43; 44; 45; 46; 47; 48; 49; 50; 51; 52; 53; 54; 55; 56; 57; 58; 59;
 60; 61; 62; 63; 64; 65; 66; 67; 68; 69; 70; 71; 72; 73; 74; 75; 76; 77; 78;
```

79; 80; 81; 82; 83; 84; 85; 86; 87; 88; 89; 90]*)

clean_up (cree 1) 90 0.01;;

(*Question 7*)

(** subset_sum_2x : int list -> int -> float -> int list

@param l Une liste des entiers , s un entier ,delta un float

@return une liste des entiers qui sont les somme des sou-ensembles mais apres nettoyage

*)

(** subset_sum_2 : int list -> int -> float -> int

@param l Une liste des entiers , s un entier ,delta un float

@return un entier qui est le plus grand somme des sous-ensembles de l mais inderieur à s

*)

(* tester s = 90, δ = 0.1 et la liste des entiers de 1 a` 100

- : int = 84*)

subset_sum_2(cree 1) 90 0.1;;

(* tester s = 90, δ = 0.01 et la liste des entiers de 1 a` 100

- : int = 90*)

subset_sum_2(cree 1) 90 0.01 ;;

(*Question 8*)

(** is_feasible : int -> int list -> int list -> bool

@param somme un entier , liste1 Une liste des entiers croissants, liste2 Une liste des entiers de´croissant

@return true si somme s'écrit comme la somme d'un élément de liste1 et d'un élément de liste2, et false sinon

*)

(* tester somme = 12 , liste1 = [1;2;4;7] , liste2 =[10;8;2]

- : bool = true*)

is_feasible 12 [1;2;4;7] [10;8;2] ;;

(* tester somme = 16 , liste1 = [1;2;4;7] , liste2 =[10;8;2]

- : bool = false*)

is_feasible 16 [1;2;4;7] [10;8;2] ;;

(*Question 9*)

(** best_feasible : int -> int list -> int list -> int

@param somme un entier , liste1 Une liste des entiers croissants, liste2 Une liste des entiers de´croissant

@return un entier qui est la somme d'un element de liste1 et d'un element de liste2, mais interieur à somme

```
*)

(* tester somme = 16 ,  liste1 = [1;2;4;7] , liste2 =[10;8;2]
- : int = 12*)
best_feasible 12 [1;2;4;7] [10;8;2] ;;
(* tester somme = 16 ,  liste1 = [1;2;4;7] , liste2 =[10;8;2]
- : int = 15*)
best_feasible 16 [1;2;4;7] [10;8;2] ;;
(* tester somme = 1000 ,  liste1 = [10;20;40;70] , liste2 =[10;8;2]
- : int = 80*)
best_feasible 1000 [10;20;40;70] [10;8;2] ;;
```

```
(*Question 10*)
```

```
(** separer : 'a list -> 'a list * 'a list
@param  l Une liste
@return l1,l2 deux listes qui sont sous-ensembles de tailles égales
*)

(* tester [10;20;40;70]
- : int list * int list = ([10; 40], [20; 70])*)

(** subset_sum_3 : int list -> int -> int
@param l Une liste des entiers ,s un entier
```

@return un entier qui est le plus grand somme des sous-ensembles de l mais inderieur à s

*)

(*Question 11*)

(** gen_random : int -> int -> int list

@param n un entier , m un entier

@return une liste contient n nombres des entiers interieur à m*)

(*tester n = 4 , m = 10

test_liste1 : int list = [1; 2; 5; 4]*)

let test_liste1=gen_random 4 10 ;;

(*Question 13*)

(** subset_sum_0 : int list -> int -> int * int list= <fun>

@param l Une liste des entiers ,s un entier

@return un couple entier * liste qui est le plus grand somme des sous-ensembles * ce sou-ensemble de l mais inderieur à s *)

(** get_all_sums : int list -> (int * int list) list

@param l Une liste des entiers

@return une liste des entiers * une liste des entiers qui cotient tous les sommes des sous-ensembles et ceux sous-ensemble de l

```
*)

ster list =[1; 3; 4; 7]
- : (int * int list) list =[(0, []); (7, [7]); (4, [4]); (11, [4; 7]); (3, [3]); (10, [3; 7]);
 (7, [3; 4]); (14, [3; 4; 7]); (1, [1]); (8, [1; 7]); (5, [1; 4]);
 (12, [1; 4; 7]); (4, [1; 3]); (11, [1; 3; 7]); (8, [1; 3; 4]);
 (15, [1; 3; 4; 7])]*)

(** subset_sum_1 : int list -> int -> int * int list
@param l Une liste des entiers ,s un entier
@return un couple entier * liste  qui est le plus grand somme des sous-ensembles * ce sou-ensemble de l mais inderieur à s
*)

(**  insert : 'a * 'b -> ('a * 'b) list -> ('a * 'b) list
@param x un couple des deux element ,l Une liste croissante contenant des couples
@return une liste croissante qui contient  le couple x
*)

(** sort : ('a * 'b) list -> ('a * 'b) list
@param l Une liste de couple
@return une liste croissante avoir les meme couples
*)


(** clean_up : (int * 'a list) list -> int -> float -> (int * 'a list) list
@param l Une liste de couple  entiers*element , s un entier ,delta un float
@return une liste de couple apres lèalgorithme nettoyage
```

*)

(** subset_sum_2x : int list -> int -> float -> (int * int list) list

@param l Une liste des entiers , s un entier ,delta un float

@return une liste de couple qui sont entiers*listes qui sont les somme des sou-ensembles mais apres nettoyage

*)

(** subset_sum_2 : int list -> int -> float -> int * int list

@param l Une liste des entiers , s un entier ,delta un float

@return un couple entier * liste qui est le plus grand somme des sous-ensembles * ce sou-ensemble de l mais inderieur à s

*)

(** best_feasible : int -> (int * 'a list) list -> (int * 'a list) list -> int * 'a list

@param somme un entier , liste1 Une liste de couple entier*liste croissants, liste2 Une liste de couple entier*liste decroissant

@return un couple entier*liste qui est la somme d'un element de liste1 et d'un element de liste2, mais interieur à somme

*)

(** subset_sum_3 : int list -> int -> int * int list

@param l Une liste des entiers ,s un entier

@return un couple entier * liste qui est le plus grand somme des sous-ensembles * ce sou-ensemble de l mais inderieur à s

*)

```
(*tester avec l=test_liste1 ,s =20
 res0 : int * int list = (12, [1; 2; 5; 4])
 val res1 : int * int list = (12, [1; 2; 5; 4])
val res2 : int * int list = (12, [1; 2; 5; 4])
val res3 : int * int list = (12, [1; 5; 2; 4])*)

let res0 = subset_sum_0 test_liste1 20;;
let res1 = subset_sum_1 test_liste1 20;;
let res2 = subset_sum_2 test_liste1 20 0.01;;
let res3 = subset_sum_3 test_liste1 20;;

(*tester avec l=test_liste2 ,s =9
val res0 : int * int list = (8, [5; 0; 3])
 val res1 : int * int list = (8, [5; 0; 3])
val res2 : int * int list = (8, [5; 3])
val res3 : int * int list = (8, [3; 5; 0])*)
let res0 = subset_sum_0 test_liste2 9;;
let res1 = subset_sum_1 test_liste2 9;;
let res2 = subset_sum_2 test_liste2 9 0.01;;
let res3 = subset_sum_3 test_liste2 9;;
```