Projet IPF: SUBSET-SUM-OPT

Romain PEREIRA

03/04/2018

Sommaire

1 Préambule

Ce projet est réalisé dans le cadre de mes études à l'ENSIIE. Rappel de l'énoncé:

> SUBSET-SUM-OPT - Etant donnée un ensemble fini E d'entiers strictements 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 $\sum_{e \in E'} e = s'$.

Ce rapport présente (en pseudo-code), les algorithmes implémentés.
Le code OCaml est disponible dans le rendu.
Un Makefile est disponible pour compiler le projet et les tests:

- *make* : compile les fichiers sources avec les tests vers un executable **subset-sum-opt.out**
- *clean* : supprimes les fichiers compilés temporaires (.mlo et .cmo)
- *fclean* : supprimes tous les fichiers compilés (executable et temporaires)

Les testes sont unitaires et vérifie, dans l'ordre:

- Les fonctions du module *Listes*
- Les fonctions du module *Ensembles*
- Les fonctions du module *Approche_naïve*

2 Spécificités techniques

2.1 Notations

J'utiliserai les notations suivantes:

- Soit E est un ensemble, $Card(E)$ est le cardinal de E.
- Soit E est un ensemble, $P(E)$ est l'ensemble formé des parties de E

2.2 Rappels/Complexités

Soit E un ensemble tel que $Card(E) = n$, alors

$$Card(P(E)) = 2^n$$

$$\forall E' \in P(E), 0 <= Card(E') <= n$$

On considèrera les complexités pour les opérations suivantes:

- Soit (x, y) des scalaires, je considère que les 4 opérations élémentaires ($x + y$, $x - y$, $x * y$, x/y) sont en $O(1)$
- Soit 2 ensembles X, Y, $X \cup Y$ est une opération en $O(Card(X) * Card(Y))$
- Soit $E \subset \mathbb{N}$, $\max E$ est une opération en $O(Card(E))$

3 Approche naïve

Cette approche consiste à determiner 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. Cette approche par 'force brute' est lourde.

3.1 Question 1 : somme sur un ensemble

Algorithm 1: Renvoie la somme des éléments de E

function $\textsc{Somme}(E' \subset \mathbb{N})$
  if $E' = \emptyset$ **then**
    return 0
  end if
  Soit $x \in E'$
  return x + $\texttt{Somme}(E' \backslash \{x\})$
end function

Complexité en $\boxed{O(n)}$, où $n = Card(E')$

3.2 Question 2 : génération des parties d'un ensemble

Algorithm 2: Renvoie l'ensemble des parties de E

function $\textsc{Sous-ensembles}(E \subset \mathbb{N})$
  if $E = \emptyset$ **then**
    return $\{\emptyset\}$
  end if
  Soit $x \in E$
  Soit $P \leftarrow \texttt{Sous} - \texttt{ensembles}(E \backslash \{x\})$
  return $P \cup \{E' \cup x \mid E' \in P\}$
end function

Complexité en $\boxed{O(2^n)}$, où $n = Card(E)$

3.3 Question 3 : résolution de SUBSET_SUM_OPT par force brute

Algorithm 3: Renvoie la réponse au problème SUBSET_SUM_OPT sur (E, s)

function $\textsc{subset\_sum}(E \subset \mathbb{N}, s \in \mathbb{N})$
  Soit $P \leftarrow \texttt{Sous} - \texttt{ensembles}(E)$
  return $\max_{E' \in P} \{s' \mid s' = \texttt{Somme}(E') \quad et \quad s' \leq s\}$
end function

Complexité en $\boxed{O(2^{n+1} n)}$, où $n = Card(E)$

4 Approche plus directe

Dans cette approche, plutot que de calculer l'ensemble des parties de E, on se propose de calculer l'ensemble des sommes atteignables en sommet sur les parties de E.

4.1 Question 4 : sommes atteignables

Algorithm 4: Renvoie l'ensemble des entiers s tels qu'il existe $E' \subseteq E$ vérifiant $\sum_{e \in E'} e = s$

```
function GET_ALL_SUMS(E ⊂ ℕ)
    if E = ∅ then
        return {0}
    end if
    Soit x ∈ E
    S ← Get_all_sums(E\{x})
    return S ∪ {x + s | s ∈ S}
end function
```

Si l'on suppose:

- $n = Card(E)$
- $m(n) = Card\{sommes \quad atteignables\} = \{s \mid \exists E' \subseteq E \mid \sum_{e \in E'} e = s\} \leq 2^n$

Alors la complexité de cet algorithme est en $\boxed{O(m(n)^2 * n)}$

- n : nombre de récursion
- $m(n)^2$: l'union

4.2 Question 5 : résolution de SUBSET_SUM_OPT

Algorithm 5: Renvoie la réponse au problème SUBSET_SUM_OPT sur (E, s)

```
function SUBSET_SUM(E ⊂ ℕ, s ∈ ℕ)
    Soit S ← Get_all_sums(E)
    return max{s' ∈ S | 0 ≤ s' ≤ s}
end function
```

La complexité de cet algorithme de résolution est donc en $\boxed{O(m(n)^2 * n)}$

5 Approche avec nettoyage

On peut réduire la complexité de l'approche précèdente en réduisant ce que l'on a noté $m(n)$ (le nombre de sommes atteignables. Dans cette approche, on se propose d'ajouter un 'filtre' sur l'algorithme qui génère les sommes.

5.1 Question 6 ; fonction clean_up

Le filtre (l'algorithme 'clean_up') est donnée dans l'énoncé. Cette fonction prends en paramètre:

- $E \subset \mathbb{N}$: l'ensemble a filtré
- $s \in \mathbb{N}$: un entier positif
- $\delta \in \mathbb{R}_+^*$: un réel positif (généralement $\ll 1$)

Cette fonction renvoie un nouvel ensemble $E' \subset E$, tel que:

- $\forall x \in E', x \leq s$
- Si l'on considère la suite croissante $(u_n)_{n\in\mathbb{N}}$ des éléments de E', $\forall n \in \mathbb{N}, u_{n+1} \geq (1+\delta)u_n$.

Autrement dit, tous les entiers strictement supérieurs à s sont supprimés, et si l'on considère 2 entiers consécutifs de l'ensemble trié x et y, tel que $x < y$, ils sont 'proches d'un rapport d'au moins $(1+\delta)$, y est supprimé. Ce filtre supprime donc les éléments de l'ensemble qui sont 'proches' l'un de l'autre (au regard de δ)

5.2 Question 7 : résolution de SUBSET_SUM_OPT

L'algorithme de résolution est également fourni dans l'énoncé. Il est le même que celui de 'l'approche plus directe' 4, sauf que lors du calcul des sommes atteignables filtrés par la fonction 'clean_up'.

Algorithm 6: Renvoie l'ensemble des entiers s tels qu'il existe $E' \subseteq E$ vérifiant $\sum\limits_{e\in E'} e = s$, passant les tests du filtre

function GET_ALL_SUMS_2($E \subset \mathbb{N}$)
    if $E = \emptyset$ **then**
        return $\{0\}$
    end if
    Soit $x \in E$
    $S \leftarrow$ `Get_all_sums_2`$(E\backslash\{x\})$
    return `clean_up`$(S \cup \{x+s \mid s \in S\})$
end function

Algorithm 7: Renvoie la réponse au problème SUBSET_SUM_OPT sur (E, s)

function SUBSET_SUM($E \subset \mathbb{N}, s \in \mathbb{N}$)
    Soit $S \leftarrow$ `Get_all_sums_2`(E)
    return $\max\{s' \in S \mid 0 \leq s' \leq s\}$
end function

La complexité de cet algorithme de résolution est donc en $\boxed{O(m'(n)^2 * n)}$, avec

$$m'(n) = Card\{sommes \quad atteignables \quad filtres\} \leq m(n) \leq 2^n$$

. **Attention** cependant, pour δ trop grand, on perd l'optimalité du résultat.

6 Approche de type *Diviser pour régner*

6.1 Question 8 : is_feasible

On nous demande ici 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 $true$ si s s'écrit comme la somme d'un élément de $l1$ et d'un élément de $l2$, et $false$ sinon.
La fonction implémenté suit l'algorithme suivant, qui se sert au maximum du fait que les listes soit triés:

Algorithm 8: Renvoie **true** si $\exists(x,y) \in l1 \times l2 \mid x+y=s$, **faux** sinon

function IS_FEASIBLE($s \in \mathbb{N}$, $l1 \in \mathbb{N}$, $l2 \in \mathbb{N}$)
    for $y \in l2$ (pris du plus grand au plus petit) **do**
        if $y > s$ **then**
            Passer au y **suivant** (car y est déjà plus grand que 's')
        end if
        for $x \in l1$ (pris du plus petit au plus grand) **do**
            if $x+y=s$ **then**
                Renvoyer true
            else if $x+y>s$ **then**
                Passer au y **suivant** (on a dépassé 's' sans l'atteindre)
            else(si $x+y<s$)
                Passer au x **suivant** (car $x+y$ est déjà plus petit que 's')
            end if
        end for
        Renvoyer faux ($\forall x \in l1, x+y<s$) on n'atteindra donc pas 's' pour un 'y' plus petit.
    end for
    Renvoyer faux (on n'a pas réussi à atteindre 's')
end function

6.2 Question 9 : best_feasible

Au regard de la Question 8 (6.1), 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 $l2$

N.B: non explicitement dis dans le sujet, mais si un tel entier s' n'existe pas, -1 est renvoyé.

Algorithm 9: Renvoie **true** si $\exists(x, y) \in l1 \times l2 \mid x + y = s$, **faux** sinon

```
function IS_FEASIBLE(s ∈ ℕ, l1 ∈ ℕ, l2 ∈ ℕ)
    s' ← −1
    for y ∈ l2 (pris du plus grand au plus petit) do
        if y > s then
            Passer au y suivant (car y est déjà plus grand que 's')
        end if
        for x ∈ l1 (pris du plus petit au plus grand) do
            if x + y = s then
                Renvoyer s' ← x + y
            else if x + y > s then
                Passer au y suivant (on a dépassé 's' sans l'atteindre)
            else(si x + y < s)
                if x + y > s' then
                    s' ← x + y
                end if
                Passer au x suivant (car x + y est déjà plus petit que 's')
            end if
        end for
        Renvoyer s' (on a trouvé s' maximum).
    end for
    Renvoyer s' (on n'a pas réussi à atteindre 's')
end function
```

6.3 Question 10 : résolution de SUBSET_SUM_OPT

7 Références

[1] 'Writting Cache-Friendly code', Gerson Robboy, Portland State University
http://web.cecs.pdx.edu/ jrb/cs201/lectures/cache.friendly.code.pdf.