

ENSIIE

RAPPORT DE PROJET D'IPF

---

# Projet individuel d'algorithme-programmation IPF

---

Romain PEREIRA

*Enseignant* : M. MOUILLERON

03/04/2018

# Projet IPF: SUBSET-SUM-OPT

Romain PEREIRA

03/04/2018

## Table des matières

<b>1</b>	<b>Préambule</b>	<b>2</b>
<b>2</b>	<b>Introduction</b>	<b>3</b>
<b>3</b>	<b>Approche naïve</b>	<b>4</b>
3.1	Question 1 : somme sur un ensemble . . . . .	4
3.2	Question 2 : génération des parties d'un ensemble . . . . .	4
3.3	Question 3 : résolution de SUBSET_SUM_OPT par force brute . . . . .	4
<b>4</b>	<b>Approche plus directe</b>	<b>5</b>
4.1	Question 4 : sommes atteignables . . . . .	5
4.2	Question 5 : résolution de SUBSET_SUM_OPT . . . . .	5
<b>5</b>	<b>Approche avec nettoyage</b>	<b>6</b>
5.1	Question 6 : fonction clean_up . . . . .	6
5.2	Question 7 : résolution de SUBSET_SUM_OPT . . . . .	6
<b>6</b>	<b>Approche de type <i>Diviser pour régner</i></b>	<b>7</b>
6.1	Question 8 : is_feasible . . . . .	7
6.2	Question 9 : best_feasible . . . . .	8
6.3	Question 10 : résolution de SUBSET_SUM_OPT . . . . .	8
<b>7</b>	<b>Comparaison des approches et améliorations</b>	<b>9</b>
7.1	Question 11/12 . . . . .	9
7.2	Question 13 : Bonus . . . . .	9
7.3	Illustration . . . . .	9
<b>8</b>	<b>Conclusion</b>	<b>12</b>
<b>9</b>	<b>Références</b>	<b>13</b>

# 1 Préambule

## Spécificités techniques

Un Makefile est disponible pour compiler le projet et les tests :

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

Les tests sont unitaires et effectués dans l'ordre :

- Les fonctions du module *Listes*
- Les fonctions du module *Ensembles*
- Les fonctions du module *Approche\_naïve*
- Les fonctions du module *Approche\_plus\_directe*
- Les fonctions du module *Approche\_avec\_nettoyage*
- Les fonctions du module *Approche\_diviser\_pour\_regner*
- Les fonctions du module *Approche\_diviser\_pour\_regner*
- Comparaisons des approches

## 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  $E$

## Rappels/Complexités

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

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

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

Les opérations suivantes ont pour complexité :

- Soit  $(x, y)$  des scalaires, 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  $O(Card(E))$

## Bonus

L'implémentation a également été légèrement plus loin que les algorithmes présentés dans ce document : elle permet de savoir sur quelle partie  $E' \subset E$  la somme  $\sum_{e \in E'} e = s$  a été atteinte.

**La 1ère implémentation (qui est plus simple, mais sans l'ensemble sur lequel les sommes sont obtenues) est disponible dans le dossier 'bkp' du rendu.**

## 2 Introduction

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 différentes approches pour répondre au problème, proposé par le sujet. Après l'étude de chaque approche, leurs performances seront comparées.

### 3 Approche naïve

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  $s$ . Cette approche par 'force brute' est lourde.

#### 3.1 Question 1 : somme sur un ensemble

---

Algorithm 1 – Renvoie la somme des éléments de  $E$

---

```
1: function SOMME( $E' \subset \mathbb{N}$ )
2:   if  $E' = \emptyset$  then
3:     return 0
4:   end if
5:   Soit  $x \in E'$ 
6:   return  $x + \text{Somme}(E' \setminus \{x\})$ 
7: end function
```

---

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

#### 3.2 Question 2 : génération des parties d'un ensemble

---

Algorithm 2 – Renvoie l'ensemble des parties de  $E$

---

```
1: function SOUS-ENSEMBLES( $E \subset \mathbb{N}$ )
2:   if  $E = \emptyset$  then
3:     return  $\{\emptyset\}$ 
4:   end if
5:   Soit  $x \in E$ 
6:   Soit  $P \leftarrow \text{Sous-ensembles}(E \setminus \{x\})$ 
7:   return  $P \cup \{E' \cup x \mid E' \in P\}$ 
8: end function
```

---

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

#### 3.3 Question 3 : résolution de SUBSET\_SUM\_OPT par force brute

---

Algorithm 3 – Renvoie la réponse au problème SUBSET\_SUM\_OPT sur  $(E, s)$

---

```
1: function SUBSET_SUM( $E \subset \mathbb{N}, s \in \mathbb{N}$ )
2:   Soit  $P \leftarrow \text{Sous-ensembles}(E)$ 
3:   return  $\max_{E' \in P} \{s' \mid s' = \text{Somme}(E') \text{ et } s' \leq s\}$ 
4: end function
```

---

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

## 4 Approche plus directe

Dans cette approche, plutôt 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  $\sum_{e \in E'} e = s$

---

```
1: function GET_ALL_SUMS( $E \subset \mathbb{N}$ )
2:   if  $E = \emptyset$  then
3:     return  $\{0\}$ 
4:   end if
5:   Soit  $x \in E$ 
6:    $S \leftarrow \text{get\_all\_sums}(E \setminus \{x\})$ 
7:   return  $S \cup \{x + s \mid s \in S\}$ 
8: end function
```

---

Si l'on suppose :

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

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

- $n$  : nombre de récursions
- $m(n)$  : l'union

**Remarque :** Cette algorithme fournit les sommes atteignables dans un ordre croissant. ( $\bullet < \bullet$ )

### 4.2 Question 5 : résolution de SUBSET\_SUM\_OPT

Une fois l'ensemble des sommes atteignables calculées, la résolution du problème devient triviale. La complexité de cet algorithme de résolution est donc en  $\boxed{O(m(n) * n)}$

---

Algorithm 5 – Renvoie la réponse au problème SUBSET\_SUM\_OPT sur  $(E, s)$

---

```
1: function SUBSET_SUM( $E \subset \mathbb{N}, s \in \mathbb{N}$ )
2:   Soit  $S \leftarrow \text{get\_all\_sums}(E)$ 
3:   return  $\max\{s' \in S \mid 0 \leq s' \leq s\}$ 
4: end function
```

---

## 5 Approche avec nettoyage

On peut réduire la complexité de l'approche précédente 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 donné dans l'énoncé. Cette fonction prend en paramètre :

- $E \subset \mathbb{N}$  : l'ensemble à filtrer
- $s \in \mathbb{N}$  : 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  $x < y$ , et s'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  $\delta$ ). En supprimant des valeurs proches, on espère pouvoir former les mêmes sommes qu'avec toutes les valeurs, mais en effectuant ainsi moins de sommations.

Par exemple, pour l'entrée  $\llbracket 0, 100 \rrbracket$ ,  $s = 90$  et  $\delta = 0.01$ , on obtient la sortie

[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]

On remarque que les valeurs les plus 'petites' sont conservées ( $\leq 10$ ), et que plus les valeurs deviennent 'grandes', moins elles passent le filtre.

### 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 filtrées par la fonction `clean_up`.

---

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

---

```
1: function GET_ALL_SUMS_2( $E \subset \mathbb{N}, \delta \in \mathbb{R}_+^*$ )
2:   if  $E = \emptyset$  then
3:     return  $\{0\}$ 
4:   end if
5:   Soit  $x \in E$ 
6:    $S \leftarrow \text{get\_all\_sums\_2}(E \setminus \{x\})$ 
7:   return clean_up( $S \cup \{x + s \mid s \in S\}$ )
8: end function
```

---

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

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

. **Attention** cependant, pour  $\delta$  trop grand, on perd l'optimalité du résultat.

---

Algorithm 7 – Renvoie la réponse au problème SUBSET\_SUM\_OPT sur (E, s)

---

```
1: function SUBSET_SUM( $E \subset \mathbb{N}, s \in \mathbb{N}$ )
2:   Soit  $S \leftarrow \text{get\_all\_sums\_2}(E, \delta)$ 
3:   return  $\max\{s' \in S \mid 0 \leq s' \leq s\}$ 
4: end function
```

---

## 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  $l_1$  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.

La fonction implémentée suit l'algorithme suivant, qui se sert au maximum du fait que les listes soient triés. Les 3 cas de retours possibles sont :

- TOO\_SMALL : Soit  $y \in l_2, \forall x \in l_1, x + y < s \Rightarrow \forall y' < y, x + y' < x + y < s$ . En français, pour  $y$  fixé, la somme des  $x + y$  avec  $x$  dans  $l_1$  est toujours inférieur à  $s$ . Pour un  $y$  plus petit, on ne risque donc pas d'atteindre une valeur plus proche de  $s$ .
- REACHED : on a réussi à atteindre  $s$ .
- TOO\_BIG : On a parcouru toutes les valeurs de  $l_2$ , pour toutes les valeurs de  $l_1$ , sans avoir pu obtenir de couple  $(x, y) \in l_1 \times l_2$  tel que  $x + y \leq s$ .

---

Algorithm 8 – Renvoie **true** si  $\exists (x, y) \in l_1 \times l_2 \mid x + y = s$ , **faux** sinon

---

```
1: function IS_FEASIBLE( $s \in \mathbb{N}, l_1 \subset \mathbb{N}, l_2 \subset \mathbb{N}$ )
2:   for  $y \in l_2$  (pris du plus grand au plus petit) do
3:     for  $x \in l_1$  (pris du plus petit au plus grand) do
4:       if  $x + y = s$  then
5:         Renvoyer true (cas REACHED)
6:       else if  $x + y > s$  then
7:         Passer au y suivant (cas TOO_BIG)
8:       else (si  $x + y < s$ )
9:         Passer au x suivant (car  $x + y$  est déjà plus petit que 's')
10:      end if
11:    end for
12:  Renvoyer faux (cas TOO_SMALL)
13: end for
14: Renvoyer faux (on a eu que des cas TOO_BIG)
15: end function
```

---

Les complexités sont :

- Dans le pire des cas (où l'on a que des cas *TOO\_BIG*) :  $O(\text{Card}(l_1)\text{Card}(l_2))$
- Dans le meilleur des cas (cas *REACHED* avec le 1er  $y$  et le 1er  $x...$ ) :  $O(1)$  (peu probable)
- Le cas moyen n'est pas évident à évaluer, étant donné que  $l_1$  et  $l_2$  dépendent de nombreux paramètres...



## 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  $l_1$  d'entier croissant, d'une liste  $l_2$  d'entier décroissant, renvoie le plus grand entier  $s' \leq s$  somme d'un élément de  $l_1$  et d'un élément de  $l_2$

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

---

Algorithm 9 – Renvoie  $\max_{(x,y) \in l_1 \times l_2} \{x + y \mid x + y < s\}$

---

```
1: function BEST_FEASIBLE( $s \in \mathbb{N}$ ,  $l_1 \subset \mathbb{N}$ ,  $l_2 \subset \mathbb{N}$ )
2:    $s' \leftarrow -1$ 
3:   for  $y \in l_2$  (pris du plus grand au plus petit) do
4:     for  $x \in l_1$  (pris du plus petit au plus grand) do
5:       if  $x + y = s$  then
6:         Renvoyer  $s$  (cas REACHED)
7:       else if  $x + y > s$  then
8:         Passer au  $y$  suivant (cas TOO_BIG)
9:       else (si  $x + y < s$ )
10:        if  $x + y > s'$  then
11:           $s' \leftarrow x + y$ 
12:        end if
13:        Passer au  $x$  suivant
14:      end if
15:    end for
16:    Renvoyer  $s'$  (cas TOO_SMALL).
17:  end for
18:  Renvoyer  $s'$  (on a eu que des cas TOO_BIG)
19: end function
```

---

## 6.3 Question 10 : résolution de SUBSET\_SUM\_OPT

En se servant des fonctions 4.1, 6.2, et en ajoutant 2 fonctions subsidiaires (voir *Listes.ml*) ...

- **split** : sépare une liste en 2 sous liste de longueur égal (à 1 prêt)
- **sort** : tri les éléments dans l'ordre croissant
- **invsort** : tri les éléments dans l'ordre décroissant

... on peut élaborer l'algorithme de résolution suivant, qui se base sur l'idée de 'diviser pour régner' :

---

Algorithm 10 – Renvoie la réponse au problème SUBSET\_SUM\_OPT sur  $(E, s)$

---

```
1: function SUBSET_SUM( $E \subset \mathbb{N}$ ,  $s \in \mathbb{N}$ )
2:    $(l_1, l_2) \leftarrow \text{split}(l)$ 
3:    $(s_1, s_2) \leftarrow (\text{get\_all\_sums}(l_1), \text{get\_all\_sums}(l_2))$ 
4:   return best_feasible( $s, \text{sort}(s_1), \text{invsort}(s_2)$ )
5: end function
```

---

## 7 Comparaison des approches et améliorations

### 7.1 Question 11/12

(voir 'tests.ml')l')

### 7.2 Question 13 : Bonus

Comme dit précédemment (1), ce bonus a été effectué : les fonctions de résolution renvoient également le sous-ensemble sur lequel la somme a été obtenu.

### 7.3 Illustration

Les graphiques suivants ont été générés sur les tests suivants :

- 1) Fixer  $n = \text{Card}(E)$
- 2) Générer un ensemble  $E \subset \mathbb{N}$  tel que  $\text{Card}(E) = n$  et  $\forall x \in E, 1 \leq x \leq 2n$
- 3) Résoudre le problème :  $\text{SUBSET\_SUM}(s = \frac{n \cdot \frac{n}{2}}{2}, E)$  avec les 4 approches.

Pour  $1 \leq n \leq 17$ , toutes les résolutions fonctionnent, et on obtient ces allures de courbes :

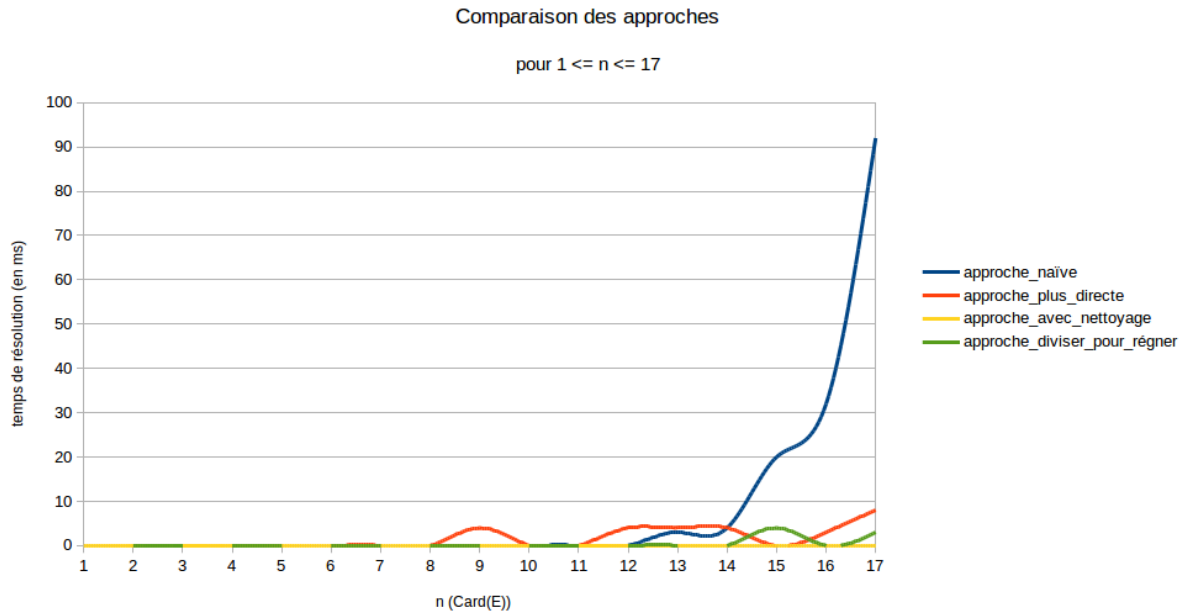


FIGURE 1 – Comparaison des approches pour  $1 \leq n \leq 17$

La complexité de l'approche naïve suit effectivement une allure exponentielle.

Pour  $n > 18$ , l'approche naïve (5) provoque une erreur de dépassement de pile, je n'ai pas pu la tester pour des  $n$  plus grands.

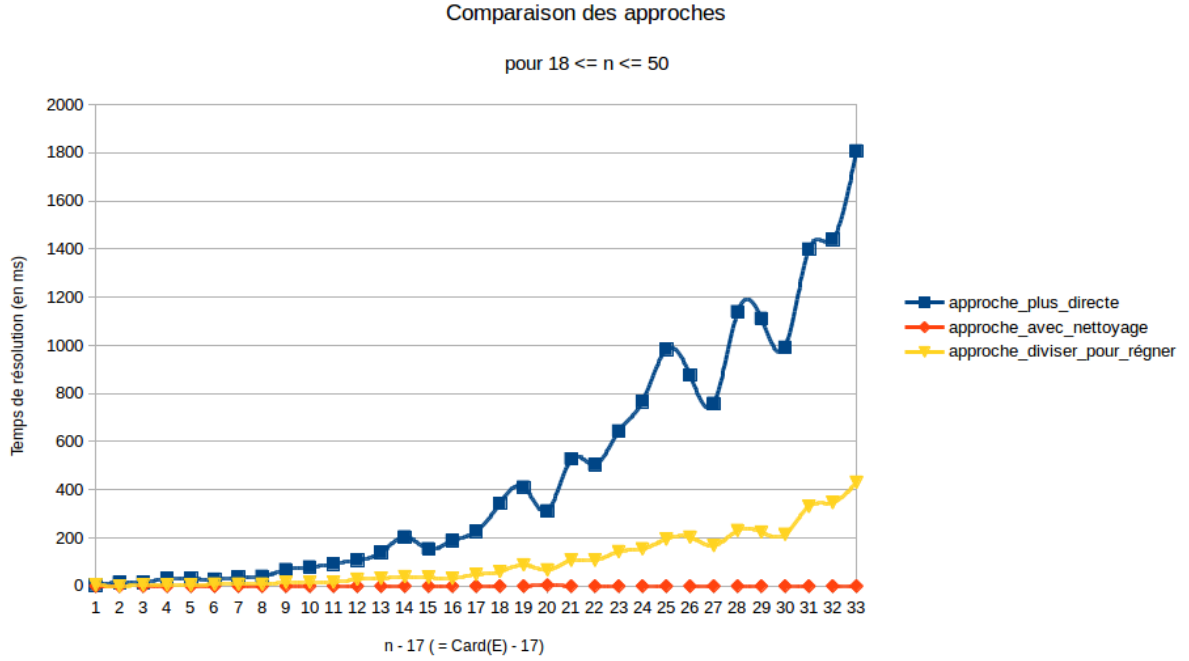


FIGURE 2 – *Comparaison des approches pour  $18 \leq n \leq 50$*

L'approche plus directe (4) a une allure exponentielle (en  $O(2^n)$ ), et que l'approche 'diviser pour régner' (6) aussi, mais avec une allure plus 'écrasé'.

Pour  $n \geq 50$ , l'approche diviser pour régner prends plus de 2 secondes... Voici les performances de l'approche avec nettoyage pour  $n \gg 1$  :

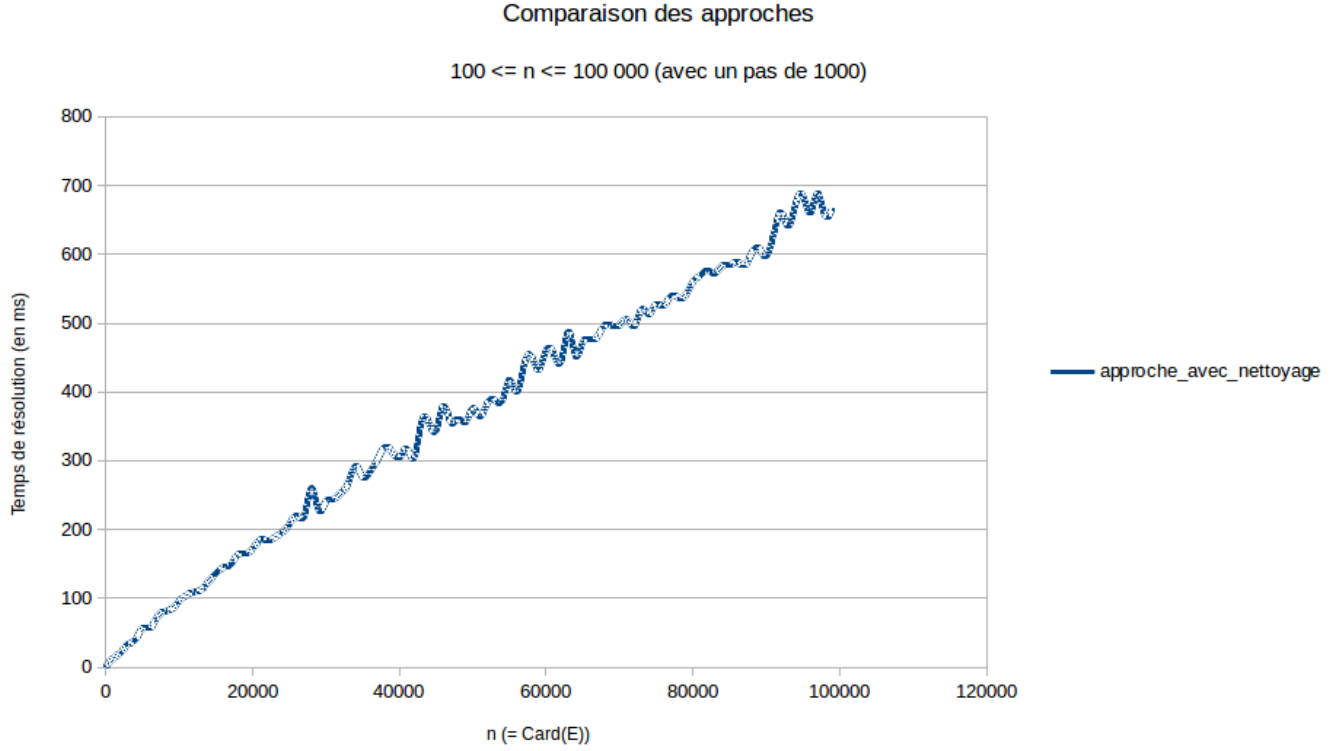


FIGURE 3 – *Comparaison des approches pour  $100 \leq n \leq 100\,000$*

Pour  $n \geq 140\,000$ , j'obtiens un dépassement de pile sur ma machine.  
Les résultats ont été obtenus pour  $\delta = 10^{-7}$ , et si  $r$  est le résultat obtenu, il s'éloigne d'au plus 1% de  $s$

$$100 \frac{r}{s} \leq 0.01$$

**N.B :** Les données brutes (format .csv) sont disponibles dans le rendu.

## 8 Conclusion

L'objectif de ce projet était de résoudre le problème SUBSET\_SUM\_OPT 2, problème NP-difficile, à l'aide de différentes approches, puis de comparer les résultats obtenus.

L'implémentation des 4 méthodes demandées a été effectuée avec succès.

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

Cependant, pour des valeurs de  $n = Card(E)$  trop grandes ( $> 50$ ), les implémentations sont lentes ( $\geq 4sec.$ ), ou pire, elles atteignent les limites de la machine et s'arrêtent suite à des dépassements de piles.

L'approche avec nettoyage quant à elle est plus rapide (voir Illustrations 7.3), mais ne permet de trouver une solution optimale.

Pour aller plus loin dans la résolution du problème, il est donc maintenant question de savoir s'il existe des structures de données permettant d'optimiser les algorithmes (l'utilisation d'une pile explicite en mémoire RAM, au lieu de la pile-machine pour éviter les dépassements de pile, par exemple), ou bien encore d'autres algorithmes plus performant donnant une solution exacte (notamment en allant plus loin dans la logique de 'diviser pour régner', ou en trouvant un filtre type 'clean\_up' qui garantit tout de même l'optimalité)

Quelques recherches ont été effectuées sur ces pistes, mais n'ont pas pu être mises en oeuvre par manque de temps.

## 9 Références

- [1] 'Module List' - INRIA  
*<https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html>*
- [2] 'Ensemble des parties d'un ensemble' - Wikipédia  
*[https://fr.wikipedia.org/wiki/Ensemble\\_des\\_parties\\_d'un\\_ensemble](https://fr.wikipedia.org/wiki/Ensemble_des_parties_d'un_ensemble)*