Ce devoir est d'une durée de 4h, et et composé de deux parties indépendantes.

1 Somme et partition de sous-ensembles

Remarque : dans cette partie, les fonctions demandées sont à écrire dans le langage C.

Le but de cette partie est d'étudier les deux problèmes de décision suivants :

PARTITION	
Entrée :	un multi-ensemble fini S d'entiers positifs
Question :	Peut-on partitionner $S = S_1 \uplus S_2$ en deux multi-ensembles S_1 et S_2 tels que $\sum_{x \in S_1} x = \sum_{x \in S_2} x$?

SUBSET-SUM	
Entrée :	un multi-ensemble fini S d'entiers positifs, un entier $t \in \mathbb{N}$
Question :	Existe-t-il $S_1 \subseteq S$ tel que $\sum_{x \in S_1} x = t$?

1.1 Difficultés des deux problèmes

1. Montrer que PARTITION $\in$ NP.
2. Montrer que SUBSET-SUM $\in$ NP.
3. Montrer que PARTITION $\leq_P$ SUBSET-SUM.
   Indication : faire deux cas, selon si $\sum_{x \in S} x$ est pair ou impair.
4. (a) Soit (S, t) une instance de SUBSET-SUM, et soit $t' = \sum_{x \in S} x - t$. Justifier que (S, t) a une solution pour le problème SUBSET-SUM si et seulement si (S, t') en a une.
   (b) Montrer que SUBSET-SUM $\leq_P$ PARTITION.
   Indication : si (S, t) est une instance de SUBSET-SUM, on pourra construire une instance S' de PARTITION en rajoutant à S une valeur dépendant de t et de $\sum_{x \in S} x$ (faire trois cas).
5. En déduire que PARTITION est NP-complet si et seulement si SUBSET-SUM est NP-complet.

1.2 Résolution et calcul d'une solution

Dans cette partie, on s'intéresse à la résolution des problèmes PARTITION et SUBSET-SUM.

1.2.1 Fonctions préliminaires

Nous allons représenter un multi-ensemble S par la structure `C` suivante :

```c
struct ensemble {
  int taille;
  int* elements;
};
typedef struct ensemble ensemble;
```

Si `S` est une variable de type `ensemble` représentant un multi-ensemble S, alors :

- `S.taille` contient le nombre d'éléments de S (comptés avec leurs multiplicités) ;
- `S.elements` pointe vers le début d'un tableau de taille `S.taille`, contenant les éléments de S.

6. Écrire une fonction `bool tous_positifs(ensemble S);` prenant en argument un multi-ensemble S et renvoyant `true` si tous les éléments de S sont positifs, et `false` sinon.
7. Écrire une fonction `int somme(ensemble S);` prenant en argument un multi-ensemble S et renvoyant la somme des éléments de S.

8. Écrire une fonction `int occurrences(ensemble S, int x);` prenant en arguments un multi-ensemble S et un entier x, et renvoyant le nombre d'occurrences de x dans S.
9. Écrire une fonction `bool est_sous_ensemble(ensemble S, ensemble S1);` prenant en arguments deux multi-ensembles S et S_1 et renvoyant `true` si $S_1 \subseteq S$, et `false` sinon.
10. Écrire une fonction `bool est_partition(ensemble S, ensemble S1, ensemble S2);` prenant en arguments trois multi-ensembles S, S_1 et S_2 et renvoyant `true` si $S = S_1 \uplus S_2$, et `false` sinon.

1.2.2 Résolution naïve

Une manière naïve de résoudre Subset-Sum serait de tester tous les sous-ensembles possibles S_1 de S jusqu'à en trouver un tel que $\sum_{x \in S_1} x = t$. De plus, plutôt que de tester absolument tous les sous-ensembles `S1` possibles, on peut se restreindre à ceux dont les éléments du tableau `S1.elements` apparaissent dans le même ordre que dans `S.elements`.

11. (a) Si `S` est un multi-ensemble de taille n, combien y a-t-il de sous-tableaux possibles de `S.elements` dont les éléments apparaissent dans le même ordre que dans `S.elements` ?
    (b) Quelle serait la complexité temporelle dans le pire des cas de cette approche naïve ?

On fournit le code de la fonction suivante :

```
bool sous_somme(ensemble S, int t, int n_max){
  if (t == 0){
    return true;
  }
  if (t < 0 || n_max < 0){
    return false;
  }
  return sous_somme(S, t - S.elements[n_max], n_max-1) || sous_somme(S, t, n_max-1);
}
```

On introduit la notation suivante : si `t` et un tableau et i et j sont deux indices, `t[i:j]` désigne les éléments de `t` dont les indices sont dans $[\![i, j-1]\!]$.

12. Montrer que la fonction `sous_somme(S,t,n_max)` termine, pour tout multi-ensemble `S` d'entiers positifs et tous entiers `n_max` $<$ `S.taille` et `t` $\in \mathbb{N}$.
13. Montrer que pour tout multi-ensemble `S` d'entiers positifs et tous entiers `n_max` $<$ `S.taille` et `t` $\in \mathbb{N}$, `sous_somme(S,t,n_max)` renvoie `true` si et seulement s'il existe un sous-tableau de `S.elements[0:n_max+1]` dont la somme des éléments vaut `t`.
14. (a) À l'aide de la fonction `sous_somme`, écrire une fonction `bool partition(ensemble S);` prenant en argument un multi-ensemble S d'entiers positifs, et renvoyant `true` s'il est possible de partitionner $S = S_1 \uplus S_2$ en deux multi-ensembles S_1 et S_2 tels que $\sum_{x \in S_1} x = \sum_{x \in S_2} x$ (et `false` sinon).
    (b) Quelle est la complexité temporelle de votre fonction `partition` dans le pire des cas ?

1.2.3 Résolution par programmation dynamique

On cherche maintenant à améliorer la complexité dans le pire des cas de la fonction `sous_somme` en utilisant de la programmation dynamique. Pour cela, si `S` est un multi-ensemble de taille n, on va construire une matrice de booléens `T` de taille $(n+1) \times (\texttt{total}+1)$ telle que :

$$\texttt{T[i][j]} = \begin{cases} \texttt{true} & \text{s'il existe un sous-ensemble de } \texttt{S.elements[0:i]} \text{ dont la somme vaut } j\,; \\ \texttt{false} & \text{sinon.} \end{cases}$$

Cette matrice `T` sera implémentée par un tableau de type `bool**`.

15. Écrire une fonction `bool** init_false(int n, int m);` prenant en arguments deux entiers positifs n et m, et renvoyant une matrice `T` de taille $n \times m$ dont toutes les cases sont initialisées à `false`.
16. Écrire une fonction `void free_matrice(bool** T, int n, int m);` libérant une telle matrice `T` (de taille $n \times m$) de la mémoire.
17. Écrire une fonction `bool sous_somme_dyna(ensemble S,int t);` prenant en entrée un multi-ensemble S d'entiers positifs et une valeur $t \in \mathbb{N}$ et renvoyant `true` s'il existe un sous-ensemble de S dont la somme des éléments vaut t, et `false` sinon.
    Attention : votre fonction devra avoir une complexité polynomiale en `S.taille` et `t`.
18. En déduire une fonction `bool partition_dyna(ensemble S);` prenant en argument un multi-ensemble S d'entiers positifs, et renvoyant `true` s'il est possible de partitionner $S = S_1 \uplus S_2$ en deux multi-ensembles S_1 et S_2 tels que $\sum_{x \in S_1} x = \sum_{x \in S_2} x$ (et `false` sinon).
    Attention : votre fonction devra avoir une complexité polynomiale en `S.taille`.

1.3 NP-complétude

Le but de cette dernière sous-partie est de montrer que les deux problèmes Partition et Subset-Sum sont NP-complets. Pour cela, on va montrer que 3-Sat $\leq_P$ Subset-Sum, où 3-Sat est le problème de décision rappelé ci-dessous.

3-Sat	
Entrée :	une formule propositionnelle φ sous forme normale conjonctive, où chaque clause est de taille au plus 3
Question :	φ est-elle satisfiable ?

Soit φ une formule au format 3-Sat, ayant n variables (notées $x_1, \dots, x_n$) et k clauses (notées $C_1, \dots, C_k$). Notre but est de construire en temps polynomial un multi-ensemble S d'entiers positifs et un entier t tels que φ soit satisfiable si et seulement si $\exists S_1 \subseteq S, \sum_{x \in S_1} x = t$.
Pour cela, nous allons construire un multi-ensemble d'entiers ayant au plus $n + k$ chiffres en base 10, chaque chiffre permettant d'encoder une partie de φ. Pour simplifier, nous supposerons que toutes ces valeurs ont $n + k$ chiffres, quitte à rajouter des 0 au début de son écriture. Dans la suite, le i-ème chiffre d'un entier désigne son i-ème chiffre en partant de la gauche dans son écriture en base 10 sur $n + k$ chiffres. Par exemple, si $n + k = 7$, le 2-ème chiffre de 0012345 est 0, son 5-ème chiffre est 3.
On construit un multi-ensemble S contenant les valeurs suivantes :

- pour chaque variable x_i, deux valeurs v_i et $\overline{v_i}$ telles que :
    - les n premiers chiffres de v_i et $\overline{v_i}$ sont tous des 0, sauf le i-ème chiffre qui est un 1 ;
    - pour $j \in [\![1, k]\!]$, le $n + j$-ème chiffre de v_i (resp. $\overline{v_i}$) vaut 1 si la clause C_j contient le littéral x_i (resp. $\neg x_i$), et 0 sinon.

  Autrement dit, pour ces valeurs, les n premiers chiffres indiquent de quel littéral on parle, et les k derniers chiffres indiquent dans quelles clauses est présent ce littéral.
- pour chaque clause C_j, deux valeurs de *padding* (qui veut dire **rembourrage** en anglais) p_j et p'_j telles que :
    - le $n + j$-ème chiffre de p_j et p'_j vaut 1 ;
    - tous les autres chiffres de p_j et p'_j valent 0.

  On a donc $p_j = p'_j$, et le multi-ensemble S contient ces valeurs en double.

À ce multi-ensemble S, on associe la valeur $t = \underbrace{1 \dots 1}_{n} \underbrace{3 \dots 3}_{k}$.

1.3.1 Étude d'un exemple

Exemple 1.1

Considérons la formule suivante :

$$\varphi = \underbrace{(x_1 \vee \neg x_2 \vee \neg x_3)}_{C_1} \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee \neg x_3)}_{C_2} \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee x_3)}_{C_3} \wedge \underbrace{(x_1 \vee x_2 \vee x_3)}_{C_4}$$

19. Pour la formule φ de l'exemple 1.1, reproduire sur votre copie et compléter le tableau ci-dessous, où chaque ligne représente une valeur décrite précédemment, et chaque case contient un chiffre.
(pas la peine de tracer les traits, utiliser les carreaux de votre copie)

	x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1							
$\overline{v_1}$							
v_2							
$\overline{v_2}$							
v_3							
$\overline{v_3}$							
p_1							
p'_1							
p_2							
p'_2							
p_3							
p'_3							
p_4							
p'_4							
t							

20. Donner une valuation satisfiant φ.
21. Donner un sous-ensemble des valeurs de S dont la somme vaut t.

1.3.2 Cas général

Soit φ une formule quelconque au format 3-SAT, soit S et t construits comme décrit précédemment à partir de φ.

22. Justifier que la construction de S et t peut se faire en temps polynomial en $|\varphi|$.

Un problème pourrait a priori perturber notre encodage : si jamais une somme de valeurs générait une retenue dans l'écriture en base 10. Tout notre encodage se retrouverait alors décalé !

23. (a) Donner une majoration de $\sum_{x \in S} x$ en fonction de n et k.
    (b) En déduire qu'il n'y aura jamais de problème de retenue.
24. Montrer que si φ est satisfiable, alors $\exists S_1 \subseteq S$, $\sum_{x \in S_1} x = t$.
25. Réciproquement, montrer que si $\exists S_1 \subseteq S$, $\sum_{x \in S_1} x = t$, alors φ est satisfiable.
26. (a) Conclure que SUBSET-SUM et PARTITION sont NP-complets.
    (b) Y a-t-il une contradiction avec les complexités demandées aux questions 17 et 18 ? Pourquoi ?

2 Décidabilité

On rappelle l'existence d'une fonction OCaml dite "**universelle**" `eval : string -> string -> string` telle que si `code_f` est le code d'une fonction OCaml `f : string -> string`, et `x` est un argument de f, alors `eval code_f x` simule l'exécution de f sur x.

2.1 Étude de quelques problèmes

On rappelle que le problème de d'arrêt est le problème de décision suivant :

ARRÊT	
Entrée :	le code `code_f` d'une fonction OCaml `f : string -> string`, une entrée `x` de `f`
Question :	est-ce que `f(x)` termine ?

27. Montrer que ARRÊT est indécidable.

Pour chacun des problèmes de décision suivants, déterminer (en le prouvant) s'ils sont décidables ou indécidables.

28.

ARRÊTBORNÉ	
Entrée :	le code `code_f` d'une fonction OCaml `f : string -> string`, une entrée `x` de `f`, un entier n
Question :	est-ce que `f(x)` termine en moins de n opérations élémentaires ?

29.

ARRÊTSELF	
Entrée :	le code `code_f` d'une fonction OCaml `f : string -> string`
Question :	est-ce que `f(code_f)` termine ?

30.

MP2ITOMPI	
Entrée :	le code `code_f` d'une fonction OCaml `f : string -> string`
Question :	est-ce que `f("MP2I")` renvoie `"MPI"` ?

31.

SACÀDOS	
Entrée :	un ensemble de n objets de poids $p_1, \dots, p_n$ et de valeurs $v_1, \dots, v_n$, un poids maximal P et une valeur minimale V
Question :	existe-t-il un sous ensemble $I \subseteq [\![1, n]\!]$ tel que $\sum_{i \in I} p_i \leq P$ et $\sum_{i \in I} v_i \geq V$?

2.2 Calcul automatique de postcondition

Définition 2.1 : formules logiques

Dans cette sous-partie, on considère des formules de la logique du premier ordre sur la signature suivante :

- $\mathcal{V} = \{e, s\}$;
- $\mathcal{C} = \{\text{chaînes de caractères}\}$;
- $\mathcal{F} = \varnothing$;
- $\mathcal{R} = \{= (2), \preceq (2)\}$.

On se restreint aux formules sans quantificateurs. Pour φ une telle formule, et ω_e, ω_s deux chaînes de caractères, on note $\varphi(\omega_e, \omega_s)$ la formule φ où toutes les occurrences de e (resp. s) ont été remplacées par ω_e (resp. ω_s).

Exemple 2.1

Si $\varphi : (e \preceq \texttt{toto}) \to (s = \texttt{tutu} \vee \bot)$, alors :

- $\varphi(\texttt{t}, \texttt{tutu}) : (\texttt{t} \preceq \texttt{toto}) \to (\texttt{tutu} = \texttt{tutu} \vee \bot)$;
- $\varphi(\texttt{to}, \texttt{to}) : (\texttt{to} \preceq \texttt{toto}) \to (\texttt{to} = \texttt{tutu} \vee \bot)$.

Définition 2.2 : formules vraies

Une fois que les valeurs de e et s sont fixées, on défini naturellement la valeur de vérité de $\varphi(\omega_e, \omega_s)$:

- pour les formules atomiques :
  - $s_1 = s_2$ est vraie si les chaînes s_1 et s_2 sont égales ;
  - $s_1 \preceq s_2$ est vraie si s_1 est un préfixe de s_2 ;
- les connecteurs logiques sont interprétés comme d'habitude.

Exemple 2.2

Dans l'exemple 2.1, $\varphi(\texttt{t}, \texttt{tutu})$ est vraie, $\varphi(\texttt{to}, \texttt{to})$ et est fausse.

Remarque 2.1 : Formules et chaînes de caractères

Pour simplifier, on suppose qu'on peut représenter une telle formule en OCaml par une chaîne de caractère représentant cette formule.

Exemple 2.3

Exemple

```
let exemple s =
  let a = "toto" in
  let b = a ^ "tutu" in
  let formule = "(e =" ^ a ^ ") ↔ (s ≼" ^ b ^ ")" in
  formule
```

Le programme ci-dessus renvoie la formule ψ suivante :

$$(e = \texttt{toto}) \leftrightarrow (s \preceq \texttt{tototutu})$$

32. On considère la formule ψ de l'exemple 2.3 ci-dessus.
    (a) Est-ce que $\psi(\texttt{toto}, \texttt{toto})$ est vraie ?
    (b) Est-ce que $\psi(\texttt{toto}, \texttt{tutu})$ est vraie ?

Définition 2.3 : postcondition valide

Soit φ une formule logique comme précédemment, et `f : string -> string` une fonction OCaml.
On dit que φ est une **postcondition valide** de f si :
pour toutes chaînes de caractères ω_e, ω_s, si $f(\omega_e)$ termine et renvoie ω_s, alors $\varphi(\omega_e, \omega_s)$ est vraie.

33. Donner une postcondition valide des fonctions OCaml suivantes :

(a)
```
let f_a e = "mpi"
```

(b)
```
let f_b e =
  let i = ref 0 in
  while (e.[!i] <> 'a') do
    incr i
  done;
String.sub e 0 !i
```

On définit maintenant le problème de décision suivant :

PostC	
Entrée :	le code `code_f` d'une fonction OCaml `f : string -> string`, une formule φ (représentée par une chaîne de caractères)
Question :	Est-ce que φ est une postcondition valide de f ?

34. Montrer que PostC est indécidable.