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

1 Somme et partition de sous-ensembles

1.1 Difficultés des deux problèmes

1. On justifie que Partition $\in$ NP avec l'ensemble de certificats suivant :
   - certificats : les multi-ensembles d'entiers ;
   - vérification : pour S une entrée du problème et S_1 un certificat :
     - on vérifie que $S_1 \subseteq S$ (en temps polynomial) ;
     - on calcule $S_2 = S \setminus S_1$ (en temps polynomial) ;
     - on vérifie que $\sum_{x \in S_1} x = \sum_{x \in S_2} x$ (en temps polynomial).

   On a bien que S possède une solution au problème de Partition si et seulement s'il existe bien un certificat S_1 de taille polynomiale en $|S|$ qui va convenir (le $S_1 \subseteq S$ de la solution).
2. On justifie de même que Subset-Sum $\in$ NP avec l'ensemble de certificats suivant :
   - certificats : les multi-ensembles d'entiers ;
   - vérification : pour (S, t) une entrée du problème et S_1 un certificat :
     - on vérifie que $S_1 \subseteq S$ (en temps polynomial) ;
     - on vérifie que $\sum_{x \in S_1} x = t$ (en temps polynomial).

   On a bien que (S, t) possède une solution au problème de Subset-Sum si et seulement s'il existe bien un certificat S_1 de taille polynomiale en $|S|$ qui va convenir (le $S_1 \subseteq S$ de la solution).
3. Montrons que Partition $\leq_{\mathsf{P}}$ Subset-Sum : soit S une instance de Partition, construisons en temps polynomial une instance (S', t') de Subset-Sum telle que S possède une solution si et seulement si (S', t') en a une :
   - si $\sum_{x \in S} x$ est impair, S n'a pas de solution au problème de Partition (car on travaille avec des entiers) : on construit alors une instance (S', t') n'ayant pas de solution pour Subset-Sum ; par exemple : $S' = S$ et $t' = 1 + \sum_{x \in S} x$ (cela se fait bien en temps polynomial).
   - si $\sum_{x \in S} x$ est pair, posons $S' = S$ et $t' = \frac{\sum_{x \in S} x}{2}$ (cela se fait bien en temps polynomial). On a bien que S possède une solution pour le problème de Partition si et seulement si (S', t') possède une solution au problème Subset-Sum (même S_1 dans les deux cas, et S_2 son complémentaire).
4. (a) On a bien que (S, t) possède une solution pour le problème Subset-Sum si et seulement si (S, t') en possède une car il suffit de passer de S_1 à $S \setminus S_1$ pour passer d'une solution d'une instance à une solution de l'autre instance.

   (b) Montrons que Subset-Sum $\leq_{\mathsf{P}}$ Partition : soit (S, t) une instance de Subset-Sum, construisons en temps polynomial une instance S' de Partition telle que (S, t) possède une solution si et seulement si S' en a une :
   - si $2t = \sum_{x \in S} x$, alors on pose $S' = S$ et dans ce cas les deux problèmes sont équivalents ;
   - si $2t > \sum_{x \in S} x$, on pose $S' = S \uplus \{x'\}$ avec $x' = 2t - \sum_{x \in S} x$.
     Remarquons que S' vérifie $\sum_{x \in S'} x = 2t$.
     - S'il existe $S_1 \subseteq S$ tel que $\sum_{x \in S_1} x = t$, alors en posant $S_2 = S' \setminus S_1$, on a bien : $\sum_{x \in S_1} x = t = \sum_{x \in S_2} x$.

- Réciproquement, s'il existe $S' = S_1 \uplus S_2$ tels que $\sum_{x \in S_1} x = \sum_{x \in S_2} x$, alors ces deux sommes valent t, et l'une des deux parties S_1 ou S_2 ne contient pas x' : c'est donc une solution au problème Subset-Sum.

- si $2t < \sum_{x \in S} x$, on pose $S' = S \uplus \{x'\}$ avec $x' = \sum_{x \in S} x - 2t$. Posons également $t' = \sum_{x \in S} x - t$. Remarquons que S' vérifie $\sum_{x \in S'} x = 2 \cdot \sum_{x \in S'} x - 2t = 2t'$.
  De plus, d'après la question 4a, (S, t) possède une solution pour le problème Subset-Sum si et seulement si (S, t') possède une solution pour le problème Subset-Sum. Le raisonnement est alors analogue au cas précédent (en raisonnant cette fois-ci sur t' au lieu de t).

Cette construction se fait bien en temps polynomial dans tous les cas, donc on a bien une réduction qui prouve que Subset-Sum $\leq_{\mathsf{P}}$ Partition.

5. - Si Partition est NP-complet (donc NP-difficile), alors comme Partition $\leq_{\mathsf{P}}$ Subset-Sum (question 3), on a que Subset-Sum est NP-difficile. De plus, Subset-Sum $\in$ NP (question 2). Donc Subset-Sum est NP-complet.
   - Si Subset-Sum est NP-complet (donc NP-difficile), alors comme Subset-Sum $\leq_{\mathsf{P}}$ Partition (question 4b), on a que Partition est NP-difficile. De plus, Partition $\in$ NP (question 1). Donc Partition est NP-complet.

1.2 Résolution et calcul d'une solution

1.2.1 Fonctions préliminaires

6.
```
bool tous_positifs(ensemble S){
  for (int i = 0; i < S.taille; i++){
    if (S.elements[i] < 0){
      return false;
    }
  }
  return true;
}
```

7.
```
int somme(ensemble S){
  int s = 0;
  for (int i = 0; i < S.taille; i++){
    s += S.elements[i];
  }
  return s;
}
```

8.
```
int occurrences(ensemble S, int x){
  int c = 0;
  for (int i = 0; i < S.taille; i++){
    if (S.elements[i] == x){
      c++;
    }
  }
  return c;
}
```

9.
```
bool est_sous_ensemble(ensemble S, ensemble S1){
  for (int i = 0; i < S1.taille; i++){
    int x = S1.elements[i];
    if (occurrences(S,x) < occurrences(S1,x)){
      return false;
    }
  }
  return true;
}
```

10.
```
bool est_partition(ensemble S, ensemble S1, ensemble S2){
  if (!est_sous_ensemble(S,S1) || !est_sous_ensemble(S,S2)){
    return false;
  }
  for (int i = 0; i < S.taille; i++){
    int x = S.elements[i];
    if (occurrences(S,x) != occurrences(S1,x) + occurrences(S2,x)){
      return false;
    }
  }
  return true;
}
```

1.2.2 Résolution naïve

11. (a) Si `S.taille` $= n$, il y a 2^n sous-ensembles `S1` de `S` dont les éléments apparaissent dans le même ordre que dans `S`.
En effet, pour chaque élément `x` de `S.elements`, on a deux choix indépendants : soit on garde `x` dans `S1.elements`, soit on ne le garde pas.

(b) Puisqu'il y aurait un nombre exponentiel de sous-ensembles `S1` à tester, et que pour chacun d'entre eux, le calcul de `somme(S1)` se fera en temps linéaire, alors dans le pire des cas la réponse est `false` et il faudra tous les tester : on aura donc une complexité temporelle exponentielle.

12. Tout d'abord, on remarque que si $0 \leq$ `n_max` $<$ `S.taille`, l'accès à `S.elements[n_max]` (ligne 8) ne produira pas d'erreur.
Ensuite, si on passe par le `return` de la ligne 8, on effectue au plus deux appels récursifs, avec `n_max`$' =$ `n_max` $- 1$. Donc la valeur de `n_max` décroit strictement à chaque appel récursif. De plus, cette valeur est entière, donc elle va atteindre une valeur strictement négative en un nombre fini d'étapes. Or la fonction `sous_somme` termine immédiatement si `n_max` < 0 (lignes 5 et 6).
Ainsi, l'appel à `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. On montre cette propriété par récurrence sur `n_max` :
- cas `n_max` < 0 : dans ce cas, `S.elements[0:n_max+1]` est vide, donc un tel sous-ensemble n'existe pas et il est correct de renvoyer `false` (lignes 5 et 6) ;
- cas `n_max` ≥ 0 :
  - si `total` < 0, comme `S` est supposé ne contenir que des entiers positifs, un tel sous-ensemble n'existe pas, et il est correct de renvoyer `false` (lignes 5 et 6) ;
  - si `total` $= 0$, le sous-ensemble vide convient, et il est correct de renvoyer `true` (lignes 2 et 3) ;
  - sinon :
    - Soit un tel sous-ensemble `S1` existe et contient la valeur `S.elements[n_max]` : dans ce cas, en enlevant cette valeur à `S1`, on obtient un sous-ensemble de `S.elements[0:n_max]` dont

la somme des éléments vaut `t-S.elements[n_max]`. Ainsi, par hypothèse de récurrence, le premier appel récursif de la ligne 8 renvoie `true`, donc `sous_somme(S,t,n_max)` renvoie `true`, ce qui est correct.

- Soit un tel sous-ensemble `S1` existe et ne contient pas la valeur `S.elements[n_max]` : dans ce cas, `S1` est un sous-ensemble de `S.elements[0:n_max]` dont la somme des éléments vaut `t`. Ainsi, par hypothèse de récurrence, le second appel récursif de la ligne 8 renvoie `true`, donc `sous_somme(S,t,n_max)` renvoie `true`, ce qui est correct.
- Soit un tel sous-ensemble n'existe pas : dans ce cas, par hypothèse de récurrence, les deux appels récursifs de la ligne 8 vont renvoyer `false`, et `sous_somme(S,t,n_max)` renvoie `false`, ce qui est correct.

14. (a)
```
bool partition(ensemble S){
  int t = somme(S);
  int n_max = S.taille - 1;
  if (t%2 == 1){
    return false;
  }
  return sous_somme(S, t/2, n_max);
}
```

(b) Dans le pire des cas, à chaque passage dans la ligne 8 de la fonction `sous_somme`, on effectuera deux appels récursifs, ce qui donnera une complexité temporelle exponentielle.
Dans ce cas, les différents appels récursifs correspondent au test de chaque sous-ensemble `S1` de `S` dont les éléments apparaissent dans le même ordre que dans `S`.

1.2.3 Résolution par programmation dynamique

15.
```
bool** init_false(int n, int m){
  bool** T = (bool**)malloc(n * sizeof(bool*));
  for (int i = 0; i < n; i++){
    T[i] = (bool*)malloc(m * sizeof(bool));
    for (int j = 0; j < m; j++){
      T[i][j] = false;
    }
  }
  return T;
}
```

16.
```
void free_matrice(bool** T, int n, int m){
  for (int i = 0; i < n; i++){
    free(T[i]);
  }
  free(T);
}
```

17.
```
bool sous_somme_dyna(ensemble S, int t){
  bool** T = init_false(S.taille+1, t+1);
  // première colonne
  for (int i = 0; i < S.taille+1; i++){
    T[i][0] = true;
  }
  for (int i = 1; i < S.taille+1; i++){
    for (int j = 1; j < t+1; j++){
      int j2 = j - S.elements[i-1];
      if (j2 < 0){
        T[i][j] = T[i-1][j];
      }
      else {
        T[i][j] = T[i-1][j] || T[i-1][j2];
      }
    }
  }
  bool res = T[S.taille][t];
  free_matrice(T, S.taille+1, t+1);
  return res;
}
```

18.
```
bool partition_dyna(ensemble S){
  int t = somme(S);
  if (t%2 == 1){
    return false;
  }
  return sous_somme_dyna(S, t/2);
}
```

1.3 NP-complétude

1.3.1 Étude d'un exemple

19.

	x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	1	0	0	1	0	0	1
$\overline{v_1}$	1	0	0	0	1	1	0
v_2	0	1	0	0	0	0	1
$\overline{v_2}$	0	1	0	1	1	1	0
v_3	0	0	1	0	0	1	1
$\overline{v_3}$	0	0	1	1	1	0	0
p_1	0	0	0	1	0	0	0
p'_1	0	0	0	1	0	0	0
p_2	0	0	0	0	1	0	0
p'_2	0	0	0	0	1	0	0
p_3	0	0	0	0	0	1	0
p'_3	0	0	0	0	0	1	0
p_4	0	0	0	0	0	0	1
p'_4	0	0	0	0	0	0	1
t	1	1	1	3	3	3	3

20. Par exemple : $\nu(x_1) = 1,\ \nu(x_2) = 0,\ \nu(x_3) = 1$.

21. On commence par le morceau représentant la valuation de la question précédente : $\{v_1, \overline{v_2}, v_3\}$.
Regardons maintenant la somme de ces lignes, pour les colonnes C_j :
- pour $j = 2$, la somme pour la colonne C_2 vaut pour l'instant 1 : on complète par $\{p_2, p_2'\}$;
- pour $j \neq 2$, la somme pour la colonne C_j vaut pour l'instant 2 : on complète à chaque fois par une seule valeur de padding : $\{p_j\}$.

Au final, on obtient $S_1 = \{v_1, \overline{v_2}, v_3, p_1, p_2, p_2', p_3, p_4\}$ dont la somme vaut bien 1113333.

1.3.2 Cas général

22. Il y a un nombre de valeurs linéaire en $n+k$, et chaque valeur peut être construite en temps polynomial en $|\varphi|$.

23. (a)
- Concernant les n premiers chiffres, il n'y a que 2 valeurs dont le chiffre vaut 1, et le chiffre vaut 0 pour toutes les autres ;
- concernant les k derniers chiffres, il n'y a qu'au plus 5 valeurs dont le chiffre vaut 1, et le chiffre vaut 0 pour toutes les autres.

Ainsi :
$$\sum_{x \in S} x \leq \underbrace{2 \cdots 2}_{n} \underbrace{5 \cdots 5}_{k}$$

(b) Pour chaque chiffre, la somme ne dépassera donc jamais 5 : pas de problème de retenue.

24. Si φ est satisfiable, considérons une valuation ν telle que $\nu \vDash \varphi$. On construit alors S_1 en lui rajoutant :
- v_i si $\nu(x_i) = 1$;
- $\overline{v_i}$ si $\nu(x_i) = 0$.

La somme vaut bien 1 pour les n premiers chiffres. Considérons maintenant les k derniers chiffres (les $n+j$-ème chiffres, pour $j \in [\![1, k]\!]$) :
- Comme chaque clause est rendue vraie, par construction des valeurs v_i et $\overline{v_i}$, chaque $n+j$-ème chiffre est ≥ 1 ;
- comme chaque clause ne contient qu'au plus 3 littéraux, au plus 3 valeurs parmi les v_i et $\overline{v_i}$ contiennent un 1 dans son $n+j$-ème chiffre, donc chaque $n+j$-ème chiffre est ≤ 3.

Ainsi, pour $j \in [\![1, k]\!]$:
- si le $n+j$-ème chiffre vaut 3, on ne rajoute aucune valeur à S_1 ;
- si le $n+j$-ème chiffre vaut 2, on rajoute une seule valeur de padding à S_1 : p_j ;
- si le $n+j$-ème chiffre vaut 1, on rajoute deux valeurs de padding à S_1 : p_j et p_j'.

Au final, on obtient bien un $S_1 \subseteq S$ dont la somme des éléments vaut t.

25. Réciproquement, si $\exists S_1 \subseteq S, \sum_{x \in S_1} x = t$, alors :
- Soit $i \in [\![1, n]\!]$. Comme le i-ème chiffre de t vaut 1, alors S_1 contient soit v_i, soit $\overline{v_i}$ (et pas les deux). On peut donc construire sans ambiguïté la valuation ν telle que
$$\nu(x_i) = \begin{cases} 1 & \text{si } v_i \in S_1 \\ 0 & \text{si } \overline{v_i} \in S_1 \end{cases}$$
- Soit $j \in [\![1, k]\!]$. Comme le $n+j$-ème chiffre de t vaut 3, et qu'il n'y a que deux valeurs de padding dont le $n+j$-ème chiffre vaut 1 (et les autres valent 0), il y a forcément au moins une valeur parmi les v_i et $\overline{v_i}$ sélectionnés dont le $n+j$-ème chiffre vaut 1. Par construction, cela veut dire que C_j contient le littéral x_i (resp. $\neg x_i$), et donc que $\nu \vDash C_j$.

Cela étant vrai pour tout $j \in [\![1, k]\!]$, on a $\nu \vDash \varphi$: donc φ est satisfiable.

26. (a) On a donc une réduction en temps polynomial qui prouve que 3-SAT $\leq_{\mathsf{P}}$ Subset-Sum. Or 3-SAT est NP-difficile, donc Subset-Sum est NP-difficile.
De plus, Subset-Sum $\in$ NP (question 2, donc Subset-Sum est NP-complet.
Et d'après la question 5, Partition est alors aussi NP-complet.

(b) Il n'y a pas de contradiction avec les complexités demandées aux questions 17 et 18, car la complexité est polynomiale en t, mais exponentielle en $\log(t)$ qui est la taille de l'entrée t.

2 Décidabilité

2.1 Étude de quelques problèmes

27. Supposons que ArrêT soit décidable par une fonction `arret : string * string -> string`. Considérons alors la fonction `paradoxe : string -> string` suivante :

```
1 let paradoxe s = arret (s,s) with
2   | false -> ()
3   | true -> while true do () done
```

Notons `code_p` le code de la fonction `paradoxe`, et considérons l'exécution de `paradoxe code_p` :

- si `paradoxe code_p` termine, alors `arret (code_p, code_p)` renvoie `true`, donc on passe dans la ligne 3 : boucle infinie. Donc `paradoxe code_p` ne termine pas...
- si `paradoxe code_p` ne termine pas, alors `arret (code_p, code_p)` renvoie `false`, donc on passe dans la ligne 2 : on ne fait rien. Donc `paradoxe code_p` termine...

Dans tous les cas, on a une contradiction, donc une telle fonction `arret` n'existe pas : ArrêT est indécidable.

28. Montrons que le problème ArrêTBorné est décidable : en effet, on peut modifier le code source `code_f` de la manière suivante :
    - on déclare une nouvelle exception : `exception Nope ;;`
    - on déclare de même une variable globale : `let compteur = ref 0 ;;`
    - on effectue une analyse syntaxique de `code_f` pour identifier chaque instruction élémentaire du code source de f ;
    - on rajoute au dessus de chacune de ces instructions le fragment de code suivant :

```
if !compteur >= n then raise Nope else incr compteur ;
(* instruction élémentaire *)
```

    - on encadre le code source ainsi obtenu comme suit :

```
try
  let _ = (* code source de f ainsi modifié *)
  in true
with
| Nope -> false
| _ -> true (* le code a terminé et levé une exception pour une autre raison *)
```

Finalement, en notant `new_code_f` le code ainsi obtenu, on peut résoudre le problème ArrêTBorné en lançant l'exécution de `eval new_code_f x` (qui termine forcément).

29. Le problème ArrêTSelf est indécidable ; en effet, montrons que ArrêT $\leq_{\text{Dec}}$ ArrêTSelf.
Soit `(code_f,x)` une instance du problème ArrêT, on construit alors le code source `code_g` suivant (calculable par la fonction `reduction`) :

```
let reduction (code_f,x) = (* renvoie un code source code_g *)
  "let g s = eval \"" ^ code_f ^ "\" \"" ^ x ^ "\""
```

Ainsi, si `g` désigne la fonction correspondant à ce code source `code_g`, on a bien que `g(code_g)` termine si et seulement si `f(x)` termine. Autrement dit, `reduction (code_f,x)` est une instance positive de ArrêTSelf si et seulement si `(code_f,x)` est une instance positive de Arret.
Ainsi, ArrêT $\leq_{\text{Dec}}$ ArrêTSelf ; et comme ArrêT est indécidable, ArrêTSelf est donc indécidable.

30. Le problème Mp2iToMpi est indécidable ; en effet, montrons que ArrêT $\leq_{\text{Dec}}$ Mp2iToMpi.
Soit `(code_f,x)` une instance du problème ArrêT, on construit alors le code source `code_g` suivant (calculable par la fonction `reduction`) :

```
let reduction (code_f,x) = (* renvoie un code source code_g *)
  "let g s =
    try
      let _ = eval \"" ^ code_f ^ "\" \"" ^ x ^ "\"
      in \"MPI\"
    with _ -> \"MPI\""
```

Ainsi, si `g` désigne la fonction correspondant à ce code source `code_g`, on a bien que `g("MP2I")` renvoie `"MPI"` si et seulement si `f(x)` termine. Autrement dit, `reduction (code_f,x)` est une instance positive de MP2ITOMPI si et seulement si `(code_f,x)` est une instance positive de ARRET. Ainsi, ARRÊT $\leq_{\text{DEC}}$ MP2ITOMPI ; et comme ARRÊT est indécidable, MP2ITOMPI est donc indécidable.

31. Le problème SACÀDOS est décidable : en effet, il suffit de tester les 2^n sous ensembles I possibles.

2.2 Calcul automatique de postcondition

32. (a) $\psi(\texttt{toto},\texttt{toto}) = (\texttt{toto} = \texttt{toto}) \leftrightarrow (\texttt{toto} \preceq \texttt{tototutu})$ est vraie.
    (b) $\psi(\texttt{toto},\texttt{tutu}) = (\texttt{toto} = \texttt{toto}) \leftrightarrow (\texttt{tutu} \preceq \texttt{tototutu})$ est fausse car `tutu` n'est pas un préfixe de `tototutu`.

33. (a) $\varphi_a(e,s) : s = \texttt{mpi}$.
    (b) $\varphi_b(e,s) : s \preceq e$.

34. Montrons que ARRÊT $\leq_{\text{DEC}}$ POSTC.
On considère la fonction `reduction : string * string -> string * string` telle que, si `code_f` est le code source d'une fonction Ocaml `f`, et `code_e` est une entrée de `f`, alors `reduction (code_f, code_e)` renvoie un couple `(code_f, formule)` où `formule` représente une formule de postcondition :

```
let reduction (code_f, code_e) =
  let formule = "(e =" ^ code_e ^ ") → ⊥" in
  code_f, formule
```

Par exemple, `reduction (code_f, "abc")` renvoie `code_f` et la chaîne de caractères représentant la formule $(e = \texttt{abc}) \to \bot$.
Supposons que POSTC soit décidable par une fonction `postc : string * string -> bool`, alors la fonction suivante décide le problème de l'arrêt :

```
let arret (code_f, code_e) =
  let code_f, formule = reduction (code_f, code_e) in
  not (postc (code_f, formule))
```

En effet, en notant <code_e> la valeur contenue par la variable `code_e` :

$$\begin{aligned}
&\texttt{arret (code_f, code_e)} \text{ renvoie } \texttt{false}\\
\iff &\texttt{postc (code_f, formule)} \text{ renvoie } \texttt{true}\\
\iff &(e = \texttt{<code_e>}) \to \bot \text{ est une postcondition valide de } \texttt{code_f}\\
\iff &\text{l'exécution de } \texttt{code_f} \text{ sur } \texttt{code_e} \text{ ne termine pas.}
\end{aligned}$$

Car, la formule $(e = \texttt{<code\_e>}) \to \bot$ n'étant jamais vraie pour une fonction ayant pris <code_e> en argument, le seul moyen pour qu'une telle formule soit une postcondition valide est que la fonction ne termine pas sur cette entrée.
Finalement, ARRÊT étant indécidable, on en déduit que POSTC est indécidable.