Pour le dernier TD BPI, nous allons nous intéresser à un problème connu comme étant “le plus simple des problèmes difficiles” (un peu de pub : nous en reparlerons dans le cours optionnel d'algorithmique avancée de seconde année).

Exercice 1 : le problème de partition

On dispose en entrée d'un multi-ensemble `E` contenant `n` entiers strictement positifs stockés dans un tableau. Pour rappel, dans un multi-ensemble une même valeur peut apparaître plusieurs fois. On souhaite partitionner `E` en deux sous-multi-ensembles `E1` et `E2` tel que la somme des éléments de `E1` soit égale à la somme des éléments de `E2`.

Question 1

!!! question “ ” Trouvez une solution pour le multi-ensemble `E = {19, 4, 7, 19, 16, 12, 7, 12, 8, 8}`.

Correction question 1

Cliquez ici pour révéler la correction.

Il n'y a que deux solutions pour ce multi-ensemble : `E1 = {19, 19, 7, 7, 4}` / `E2 = {16, 12, 12, 8, 8}` et la partition “inverse” `E2 = {19, 19, 7, 7, 4}` / `E1 = {16, 12, 12, 8, 8}`.

Question 2

!!! question “ ” Proposez une fonction récursive renvoyant le nombre de partitions valides différentes en pensant à bien définir ce que “différentes” signifie.

Correction question 2

Cliquez ici pour révéler la correction.

La notion de partitions différentes que nous utilisons se base sur la position des éléments dans le multi-ensemble `e` et non sur la valeur.

Voici une première solution :

```
def compte_partitions_valides_rec_v1(entiers):
    """Renvoie le nombre de partitions valides différentes.

    Valide signifie que la somme des éléments du premier
    sous-multi-ensemble est égale à la somme des éléments
    du deuxième sous-multi-ensemble.

    La notion de partitions différentes se base ici sur la
    position des éléments dans le multi-ensemble `entiers`
    et non sur la valeur.
```

```
L'idée de l'implémentation est d'essayer toutes les possibilités
en plaçant chaque élément soit dans e1 soit dans e2.
"""

def _compte_partitions_valides_rec(entiers, cible):
    """Ici nous utilisons une fonction interne.

    L'objectif de cette fonction interne est de cacher
    les paramètres propres à l'implémentation à l'utilisateur
    de la fonction `compte_partitions_valides_rec`.

    `entiers` contient les éléments qu'il reste à placer.
    `cible` est l'objectif de somme à atteindre pour e2.

    `entiers` est modifié au cours des appels récursifs (sa taille
    diminue), mais reste inchangé en sortie de cette fonction.
    """

    # Si on a placé tout le monde
    if not entiers:
        return int(cible == 0)

    # On peut s'arréter si on a pas les reins
    # assez solides pour aller au bout
    if sum(entiers) < cible:
        return 0

    # On peut aussi s'arréter si on est allé trop loin
    if cible < 0:
        return 0

    # Sinon on va essayer de placer le dernier
    # élément de entiers soit dans e1 soit dans e2.
    # On prend le dernier car pop() est gratuit
    # contrairement à pop(0) (si on prenait le
    # premier).
    dernier = entiers.pop()

    # On met `dernier` dans e1 et on fait un appel récursif.
    # Comme cible est la cible pour e2, on ne la change pas
    # ici.
    res = _compte_partitions_valides_rec(entiers, cible)

    # On met `dernier` dans e2 et on fait un appel récursif
    res += _compte_partitions_valides_rec(entiers, cible - dernier)
```

```
            # NE PAS OUBLIER de remettre dernier dans entiers.
            # Là encore, comme on rajoute à la fin, c'est
            # gratuit.
            entiers.append(dernier)

            return res

    # Si la somme est impaire, on peut s'arréter
    # tout de suite.
    total = sum(entiers)
    if total % 2 != 0:
        return 0

    # Sinon, on y va !
    return _compte_partitions_valides_rec(entiers, total // 2)
```

Et voici une deuxième solution :

```
def compte_partitions_valides_rec_v2(entiers):
    """Renvoie le nombre de partitions valides différentes.

    Valide signifie que la somme des éléments du premier
    sous-multi-ensemble est égale à la somme des éléments
    du deuxième sous-multi-ensemble.

    La notion de partitions différentes se base ici sur la
    position des éléments dans le multi-ensemble `entiers`
    et non sur la valeur.

    L'idée de l'implémentation est la même que dans la version 1,
    à savoir essayer toutes les possibilités en plaçant chaque
    élément soit dans e1 soit dans e2.

    On rajoute nénamoins une optimisation qui ne change pas la
    complexité mais divise par 2 le nombre d'appels récursifs
    en ne testant pas une partition et "son symétrique" mais seulement
    l'une des deux. Si entiers = {1, 2, 3} alors e1 = {1}, e2 = {2, 3}
    est le symétrique de e1 = {2, 3}, e2 = {1}
    """

    def _compte_partitions_valides_rec(indice, cible):
        """Ici nous utilisons une fonction interne.

        L'objectif de cette fonction interne est de cacher
        les paramètres propres à l'implémentation à l'utilisateur
        de la fonction `compte_partitions_valides_rec`.
```

```
        `entiers` n'est plus passé en paramètre car on peut y accéder
        directement dans cette fonction puisqu'elle est incluse
        dans la fonction ayant `entiers` comme paramètre.
        `indice` est l'indice dans `entiers` du prochain élément à placer.
        `cible` est l'objectif de somme à atteindre pour e2.
        """

        # Si on a placé tout le monde
        if indice == len(entiers):
            return int(cible == 0)

        # On peut s'arréter si on a pas les reins
        # assez solides pour aller au bout
        if sum(entiers[indice:]) < cible:
            return 0

        # On peut aussi s'arréter si on est allé trop loin
        if cible < 0:
            return 0

        # Sinon on va essayer de placer `e[indice]`
        # soit dans e1 soit dans e2.

        # On met `e[indice]` dans e1 et on fait un appel récursif.
        # Comme cible est la cible pour e2, on ne la change pas
        # ici.
        res = _compte_partitions_valides_rec(indice + 1, cible)

        # On met `e[indice]` dans e2 et on fait un appel récursif
        res += _compte_partitions_valides_rec(indice + 1, cible - entiers[indice])

        return res

    # Si la somme est impaire, on peut s'arréter
    # tout de suite.
    total = sum(entiers)
    if total % 2 != 0:
        return 0

    # Sinon, on y va !

    # L'ensemble vide doit  être traité séparément car
    # dans ce qui suit on commence au premier élément de e
    if not entiers:
        return 1
```

```
    # Pour ne pas tester les partitions symétriques, on
    # met e[0] dans e1 et on compte deux fois chaque
    # solution.
    return 2 * _compte_partitions_valides_rec(1, total // 2)
```

Question 3

!!! question " " Proposez une fonction itérative renvoyant le nombre de partitions valides différentes.

Correction question 3

Cliquez ici pour révéler la correction.

Voici une solution itérative utilisant `itertools.combinations` :

```
def compte_partitions_valides_iter(entiers, affiche=False):
    """Renvoie le nombre de partitions valides différentes.

    Valide signifie que la somme des éléments du premier
    sous-multi-ensemble est égale à la somme des éléments
    du deuxième sous-multi-ensemble.

    La notion de partitions différentes se base ici sur la
    position des éléments dans le multi-ensemble `entiers`
    et non sur la valeur.

    L'idée ici est d'utiliser le module itertools pour
    récupérer toutes les combinaisons possibles de toutes
    les tailles possibles pour `e2` et de vérifier si leur somme
    est égale à la moitié de la somme des éléments de `entiers`.

    ATTENTION dans cette solution il n'y a pas de "coupe" contrairement
    aux solutions récursives car `e2` n'est pas construit progressivement.
    Cette solution risque donc d'être moins intéressante quand des coupes
    sont possibles.
    """

    # L'ensemble vide est un cas particulier
    if not entiers:
        return 1

    # Si la somme est impaire, on peut s'arrêter
    # tout de suite.
    total = sum(entiers)
    if total % 2 != 0:
```

```
        return 0

    count = 0
    cible = total // 2
    # Pour ne pas tester les symétriques on enlève
    # le dernier et on fera x 2 au final
    dernier = entiers.pop()
    for multiset_2_size in range(1, len(entiers)):
        combinations = itertools.combinations(entiers, multiset_2_size)
        for multiset_2 in combinations:
            if sum(multiset_2) == cible:
                if affiche:
                    print(" ", multiset_2)
                count += 1
    entiers.append(dernier)
    return 2 * count
```

Exercice 2 : n-reines (pour aller plus loin)

On considère un échiquier de taille `n x n`. On rappelle qu'une reine aux échecs menace toutes les cases situées sur les mêmes lignes, colonnes ou diagonales qu'elle. On souhaite placer `n` reines sur l'échiquier de taille `n x n` sans qu'elles se menacent entre elles.

Question 1

!!! question " " Proposez une fonction récursive calculant le nombre de placement différents des `n` reines répondant à la contrainte ci-dessus.

Correction question 1

Cliquez ici pour révéler la correction.

```
#!/usr/bin/env python3

"""Le problème des n reines."""

import time


def conflict(queen1, queen2):
    """Return if the two given queens are in conflict."""
    if queen1[0] == queen2[0]:  # on the same line (should never happen in our case)
        assert False, "two queens on the same line !"
        # return True
    elif queen1[1] == queen2[1]:  # on the same column
        return True
```

```
    elif abs(queen1[0] - queen2[0]) == abs(
        queen1[1] - queen2[1]
    ):  # on the same diagonal
        return True
    return False


def has_conflict(queens):
    """Return if the last queen of queens is in conflict with at least one other queen."""
    last_queen = queens[-1]
    for queen in queens[:-1]:
        if conflict(last_queen, queen):
            return True
    return False


def nb_valid_configs_params(size):
    """Return the number of valid configurations with 'size' queens for a boardgame sizexsi

    def nb_valid_configs_params_rec(queens, size):
        """Internal recursive function to compute the number of valid configurations

        - queens is a list of queens already on the boardgame.
          A queen is a tuple (line_no, col_no).
          queens contains a single queen per line, since it's useless to try to put more
          and because we add queens line by line, queens[i][0] == i is *always* True
        - size is the size of one side of the boardgame
        """

        # Stop the recursion if we sucessfully added a queen on every line
        if len(queens) == size:
            return 1

        # Else we make a recursive call for each possible column on the current line
        count = 0
        current_line = len(queens)
        for column in range(size):
            queens.append((current_line, column))
            # Stop the recursion if at least two queens are in conflict
            # and return 0 meaning this branch of the recursion tree has 0
            # valid configurations
            if not has_conflict(queens):
                count += nb_valid_configs_params_rec(queens, size)
            queens.pop()

        return count
```

```
    queens = []  # We start with an empty boardgame
    return nb_valid_configs_params_rec(queens, size)


def test_functions():
    """Test all functions above"""

    sizes = range(1, 12)
    for size in sizes:
        start = time.process_time_ns()
        nb_valid_configs = nb_valid_configs_params(size)
        end = time.process_time_ns()
        size_s = f"{size}x{size}"
        print(
            f"size {size_s:5} --> {nb_valid_configs:4d} "
            f"configs in {(end - start) * 1E-9:.3f} seconds"
        )


if __name__ == "__main__":
    test_functions()
```