# Mémoïsation

## 1 Mots de taille n

On considère dans cette partie les entiers naturels écrits en binaires sur $n$ bits. L'objectif est de déterminer combien de mots ne contiennent pas deux 0 consécutifs dans leur écriture.

On formalise le problème algorithmique pour tout entrée $n \in \mathbb{N}$ :

$$P_n = |\{0 \leq k < 2^n,\ k \text{ ne possède pas 2 zéros consécutif dans sont écriture en binaire}\}|$$

**Exercice 1.** *Étude théorique du problème*

1. Décrivez une solution pour calculer $P_n$ en utilisant une recherche exhaustive de solutions. Quelle est la complexité de cette solution ?
2. Quel est le type de problème algorithmique à résoudre ?
3. Exprimez $P_1$, $P_2$ et $P_n$ en fonction de $P_{n-1}$ et $P_{n-2}$ pour $n \geq 3$
4. Décrivez une solution pour calculer $P_n$ en utilisant une approche descendante.
   (a) Quelle est la complexité sans utiliser de technique de mémoïsation ?
   (b) Quelle est la complexité en utilisant une technique de mémoïsation ?

**Exercice 2.** *Implémentation de solution*

1. Implémenter une fonction `nb_c` qui prend en entrée un entier positif $n$ et renvoie $P_n$ en utilisant une approche par sous-problème descendante. Vous utiliserez une technique de mémoïsation, et choisirez la structure la plus petite et la plus efficace pour stocker les sous-problèmes calculés.

## 2 Somme de sous-ensembles

On considère dans cette partie une liste d'entiers positifs `l` et un entier `k`. L'objectif est de déterminer si il existe une liste extraite de `l` (sous-ensemble d'éléments de `l`) dont la somme vaut `k`.

On formalise le problème algorithmique pour tout liste d'entiers positifs $l$ et tout $k \in \mathbb{Z}$ :

$$P_k(l) = \exists l' \subseteq l, \texttt{sum}(l') = k$$

**Exercice 3.** *Étude théorique du problème*

1. Décrivez une solution pour calculer $P_k(l)$ en utilisant une recherche exhaustive de solutions. Quelle est la complexité de cette solution ?
2. Quel est le type de problème algorithmique à résoudre ?





3. Exprimez $P_0(l)$, $P_k(l)$ pour $k < 0$ et $P_k([])$.
4. Pour $l$ non vide, exprimez $P_k(l)$ en fonction de $l[0]$, $l[1:]$ et $k$.

**Exercice 4.** *Implémentation Descendante*

Implémenter une fonction `subset_sum_desc` qui prend en entrée une liste d'entiers positifs et un entier $k$ et renvoie $P_k(l)$ en utilisant une approche par sous-problème descendante. Sans utiliser de technique de mémoïsation.

Quelle est la complexité de votre fonction ?

**Exercice 5.** *Implémentation Descendante avec Mémoïsation par Dictionnaire*

Implémenter une fonction `subset_sum_dict` qui prend en entrée une liste d'entiers positifs et un entier $k$ et renvoie $P_k(l)$ en utilisant une approche par sous-problème descendante. Vous utiliser un dictionnaire pour mémoïser les résultats des sous-problèmes.

Quelle est la complexité de votre fonction ?

**Exercice 6.** *Implémentation Montante*

Determinez, à $k$ fixé quels sont les sous-problème intervenant dans la résolution du problème $P_k(l)$, $l$ prenant toutes les listes d'entiers possibles.

Implémenter une fonction `subset_sum_mot` qui prend en entrée une liste d'entiers positifs et un entier $k$ et renvoie $P_k(l)$ en utilisant une approche par sous-problème montante.

Prouvez la terminaison et la correction de votre fonction.

On utilisera les matrices `numpy` dont voici un exemple d'utilisation. On considère le module `numpy` importé avec pour alias `np`.

```
matrice = np.full((n,m), True) # Création de matrice n x m remplie de True
matrice[i][j] = False # Modification
x = matrice[i][j] # Accès
```

**Exercice 7.** *Implémentation Descendante avec Mémoïsation par Matrices*

Implémenter une fonction `subset_sum_mat` qui prend en entrée une liste d'entiers positifs et un entier $k$ et renvoie $P_k(l)$ en utilisant une approche par sous-problème descendante. Vous utiliser une matrice pour mémoïser les résultats des sous-problèmes.

## 3 Analyse des performances

On veux mesurer les performances des différentes fonctions `subset_sum` au pire cas en fonction de leur implémentation. Le pire cas est atteint lorsqu'il n'existe pas de liste extraite dont la somme est k.

**Exercice 8.** *Génération de jeux de test*

Implémenter une fonction `rand_subset_sum_dict` qui prend en entrée un entier `n`, génère une liste de `n` nombres aléatoires entre 0 et 99 inclus, et lance la fonction `subset_sum_dict` sur la liste de taille n générée et `k = 100*n`.

On utilisera le module `random` dont voici un exemple d'utilisation.

```
n = random.randint(3,12) # génère un entier entre 3 et 12 inclus.
```

**Exercice 9.** *Temps d'exécution moyen*

Implémenter une fonction `timing_subset_sum_dict` qui prend en entrée un entier `n` et mesure la moyenne des temps d'exécution de la fonction `rand_subset_sum_dict` appelée sur l'entrée `n` 20 fois.

Pour calculer le temps d'exécution, on utilisera le module `time`. Chaque processus possède son horloge, indiquant le temps consacré par le CPU à son exécution. La performance d'un morceau de code peut être calculé en faisant la différences de la valeur d'horloge avant et après l'exécution du morceau de code en question.

**Exercice 10.** *Courbes*

Créer une liste `courbe_subset_sum_dict` de taille 100 qui contient à l'indice `i` la valeur de la fonction `timing_subset_sum_dict(i)`.

**Exercice 11.** *Tracé*

Utiliser le module `matplotlib` pour tracer la courbe obtenue. Puis avec une méthode analogue ajoutez au tracé les courbes de performances des fonctions `subset_sum_mat`, `subset_sum_mont` et `subset_sum_desc`. Pour cette dernière on veillera à faire le tracé uniquement pour des valeurs de `n` faibles ($n \leq 7$).

```
import matplotlib.pyplot as plt

x = [ 2 / t for t in range(1,1200)] # Valeurs en absisses
y = [ t**2 for t in x ]
z = [ t**(0.5) for t in x ]
plt.plot(x,y, label="Parabole")
plt.plot(x,z, label="Racine")
plt.legend()
plt.show()
```