Université d'Artois — ALGO 4

Faculté des Sciences Jean Perrin — **TP 3** — Semestre pair 2024–2025

Exercice 1 : Un kangourou démarre à la position 1 et doit atteindre une position k, avec $k \geq 3$ en sautant (vous l'avez bien imaginer 🦘). Ses sauts sont calculés en additionnant deux distances qu'il a déjà parcourues. Ces distances sont représentées dans une liste, initialement définie par `l=[1]`.

Au départ, le kangourou ne peut effectuer qu'un saut de 1, ainsi il peut effectuer un saut de $1 + 1 = 2$, ce qui le place à la position 2. La liste des distances est alors mise à jour pour devenir `[1,2]`.

Lors de l'étape suivante, le kangourou peut combiner deux autres distances disponibles dans la liste. Par exemple :

— En combinant $1 + 2 = 3$, il atteint la position 3, et la liste devient `[1,2,3]`.

— En combinant $2 + 2 = 4$, il atteint la position 4, et la liste devient `[1,2,4]`.

Le dernier élément de la liste représente la position finale du kangourou. Les éléments de la liste indiquent les distances qui peuvent être utilisées pour de futurs sauts.

À chaque étape, le kangourou continue de combiner deux distances i et j disponibles dans la liste `l` pour effectuer un saut de distance `l[i]+l[j]`. Il répète cette stratégie jusqu'à atteindre la position finale k.

Soit la fonction définie de la manière suivante :

```python
def sauter(i:int,j:int,l=[1]):
    l.append(l[i]+l[j])
```

Afin d'aider le kangourou à optimiser sa stratégie de sauts, vous disposez de la fonction `sauter`. Cette fonction met à jour la liste des sauts du kangourou en ajoutant un $i + j$ comme dernier élément de la liste.

La stratégie naïve, que vous pourriez proposer au kangourou (si vous n'aimez pas particulièrement les kangourous 🦘), serait de sauter d'un pas de 1 à chaque fois. Cela revient à appeler la fonction `sauter` pour ajouter un 1 à chaque étape.

Le nombre d'appels à la fonction `sauter` correspond au nombre d'éléments de la liste moins un une fois arrivé à la position k.

```python
def strategie_naive(k:int):
    l=[1]
    for i in range(k-1):
        sauter(0,i,l)
    return len(l)-1
```

1. Implémenter cette stratégie naïve, puis tester-la pour les valeurs suivantes : $k = 5, 15, 199$.
2. Déterminer le nombre de sauts en fonction de k.

Une stratégie plus efficace afin de réduire le nombre de sauts, repose sur le calcul de la plus grande puissance 2^p telle que $2^p \leq k$. Si l'égalité est vérifiée, il suffit alors de sauter $2, 4, \ldots, 2^p$ jusqu'à atteindre k. Sinon, on utilise ces sauts déjà présents dans la liste jusqu'à ce que k soit atteint. Cette méthode est appelée *stratégie binaire*. Par exemple pour $k = 15$:

```
sauter(0,0,[1])  ↪  [1,2]
sauter(1,1,[1,2])  ↪  [1,2,4]
sauter(2,2,[1,2,3])  ↪  [1,2,4,8]
sauter(3,2,[1,2,4,8])  ↪  [1,2,4,8,12]
sauter(4,1,[1,2,4,8,12])  ↪  [1,2,4,8,12,14]
sauter(5,1,[1,2,4,8,12,14])  ↪  [1,2,4,8,12,14,15]
```

1. Écrire une fonction `strategie_bin` qui implémente la stratégie binaire. Effectuez les mêmes tests que précédemment pour les valeurs $k = 5, 15, 199$. Comparez les résultats obtenus avec ceux de la stratégie naïve.
2. Quel est le nombre de sauts nécessaires pour atteindre l'objectif en suivant cette stratégie en fonction de k ?

Cette stratégie n'est pas optimale et il est possible de réduire encore davantage le nombre de sauts. Par exemple, pour $k = 15$ et en effectuant les sauts comme suit, on évite un mouvement supplémentaire par rapport à la stratégie binaire :

```
sauter(0,0,[1])  ↪  [1,2]
sauter(1,0,[1,2])  ↪  [1,2,3]
sauter(2,2,[1,2,3])  ↪  [1,2,3,6]
sauter(4,4,[1,2,3,6])  ↪  [1,2,3,6,12]
sauter(5,2,[1,2,3,6,12])  ↪  [1,2,3,6,12,15]
```

Soit m_k le nombre minimum de sauts nécessaires pour atteindre la position k. Par exemple, comme on vient de le voir $m_{15} = 5$.

1. Écrire une fonction récursive qui permet de calculer la somme suivante :

$$\sum_{k=3}^{199} m_k$$

la valeur de $\sum_{k=3}^{99} m_k = 655$ cette valeur vous permettra de vérifier le bon fonctionnement de votre fonction.