Mémoïsation

I Mots de taille n

On considère dans cette partie les entiers naturels écrits en binaires sur n bits. L'*écriture* d'un entier est une liste d'entiers avec le bit de poids fort en dernière position. Par exemple `[1,0,1,1]` pour `13`.

Par convention `0` s'écrit `[]` sur 0 bits, et est le seul entier à s'écrire sur 0 bits.

L'objectif est de déterminer combien d'entiers ne contiennent pas deux 0 consécutifs dans leur écriture binaire.

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

$$\mathrm{E}_n = \{0 \leq k < 2^n, k \text{ ne possède pas 2 zéros consécutif dans sont écriture en binaire sur } n \text{ bits}\}$$

$$\mathrm{P}(n) = |\mathrm{E}_n|$$

Exercice 1. *Étude théorique du problème*

1. Quel est le type de problème algorithmique à résoudre ?
2. Décrivez une solution pour calculer P_n, en utilisant une recherche exhaustive de solutions. Quelle est la complexité de cette solution ?
3. Exprimez $\mathrm{P}(0)$, $\mathrm{P}(1)$, $\mathrm{P}(2)$ et $\mathrm{P}(n)$ en fonction de $\mathrm{P}(n-1)$ et $\mathrm{P}(n-2)$ pour $n \geq 2$. On pourra commencer par trouver une relation entre E_n, E_{n-1} et E_{n-2}
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*

Implémenter une fonction `nb_c` qui prend en entrée un entier positif n et renvoie $\mathrm{P}(n)$ en utilisant une approche par sous-problème descendante. Vous utiliserez une technique de mémoïsation.

II 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 toute liste d'entiers positifs l et tout $k \in \mathbb{Z}$:

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

Ainsi que les sous-problèmes :

$$\mathrm{P}(k, l, i) = \mathcal{P}(k, l[:i])$$

Ainsi, $\mathcal{P}(k, l) = \mathrm{P}(k, l, \texttt{len}(l))$.

Exercice 3. *Étude théorique du problème*

1. Quel est le type de problème algorithmique à résoudre ?
2. Décrivez une solution pour calculer $\mathcal{P}(k, l)$ en utilisant une recherche exhaustive de solutions. Quelle est la complexité de cette solution ?
3. Exprimez `P(0,l,i)`, `P(k,l,i)` pour $k < 0$ et `P(k,l,0)` pour $k > 0$.
4. Pour `l` non vide et $0 \leq i < n$, exprimez `P(k,l,i+1)` en fonction de `P(k,l,i)`.

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 $\mathrm{P}(k, l)$ 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 $\mathcal{P}(k, l)$ 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 et l fixés, un sous-ensemble de sous-problèmes suffisants et facile à construire (temps constant) pour la résolution du problème $\mathcal{P}(k, l)$.

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

III 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` qui prend en entrée un entier `n` et une fonction `subset_sum` puis génère une liste de `n` nombres aléatoires entre 0 et 99 inclus, et lance la fonction `subset_sum` 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` qui prend en entrée un entier `n`, une fonction `subset_sum` et mesure la moyenne des temps d'exécution de la fonction `rand_subset_sum` 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 différence de la valeur d'horloge avant et après l'exécution du morceau de code en question.

```
t = time.process_time() # Permet de sauvegarder la valeur de l'horloge du processus.
```

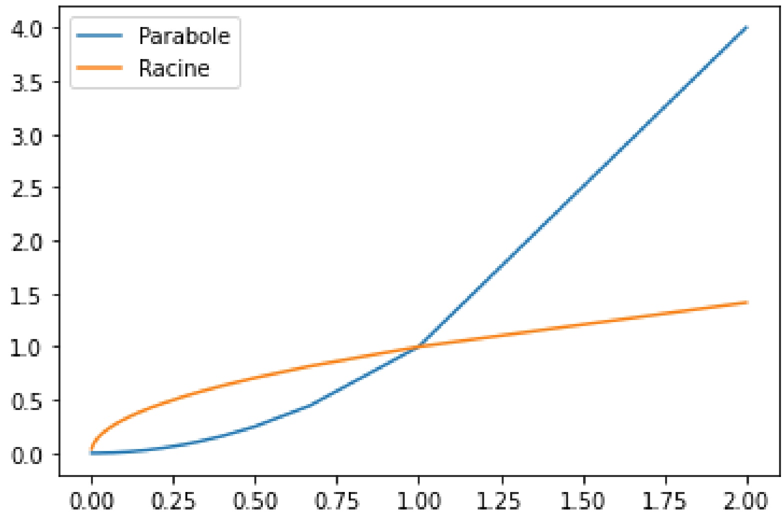
Exercice 10. *Tracé*

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

Utiliser le module `matplotlib` pour tracer la courbe des performances de chaque fonctions codées, `subset_sum_dict`, `subset_sum_mat`, `subset_sum_mont` et `subset_sum_desc`. Plus spécifiquement on pourra tracer pour chaque fonction `f` citée, les valeurs de `timing_subset_sum(i,f)` pour chaque valeur de `i` allant de 0 à 100. Pour la fonction `subset_sum_desc` on prendra uniquement les valeurs de `i` entre 0 et 7, la fonction prenant trop de temps sinon.

```
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()
```


Parabole
Racine
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00