TD/TP - Piles, mémoire, processus

1 Partie 1: TD

Exercice 1:

Quand les pingouins glissent plus loin...

Faire glisser des pingouins, c'est quand même super cool, les faire glisser plus longtemps, c'est encore mieux. Dans cet exercice, nous ne contraignons plus la récursion sur le trajet en demandant $n <= 3$ pour continuer la récursion, seulement pour l'affichage. Cela permet de calculer des trajectoires plus longues (code ci-dessous).

Listing 1: Trajectoire du Pingouin

```
def move(n: int):
    if n == 0:
        print("<", end = "␣")
    if n == 1:
        print("^", end = "␣")
    if n == 2:
        print(">", end = "␣")
    if n == 3:
        print("v", end = "␣")

def trajectory(n: int):
    if n >= 0:
        trajectory(n - 1)
        trajectory(n - 2)
        if n <= 3:
            move(n)
```

Pour compenser la difficulté associée, nous proposons une méthode systématique pour calculer la trajectoire du pingouin. Nous allons définir les "règles" suivantes :

- 0 donne ←
- 1 donne 0 ↑
- 2 donne 1 0 →
- 3 donne 2 1 ↓

On remarque que le chiffre de gauche correspond à la valeur d'appel et qu'il "donne" lors de l'appel, d'autres appels et un déplacement (une flèche, si possible). On sait que l'on peut maintenant appeler trajet avec des valeurs supérieures à trois. Donnez les "règles" pour les valeurs 4, 5 et 6 (ce qu'elles "donnent").

Exercice 2:

...Une méthode systématique est bienvenue

Nous avons vu le principe de la pile d'exécution en cours, nous allons la simuler avec un tableau

- Chaque colonne correspond à un état de pile au cours du temps (haut de pile en haut de colonne), et une colonne à droite correspond à l'état suivant de la pile.
- Si on décide de commencer la récursion avec $n = 3$, on commence par empiler "3".
- On commence ensuite la récursion :
  - Si le haut de pile est un **nombre** , on dépile et on empile ce qu'il "donne" grâce aux règles définies plus haut (on empile de droite à gauche, e.g. pour "3 donne 2 1 ↓", on empile d'abord "↓", puis "1", puis "2" (pourquoi: le haut de pile correspond à ce qui est executé en premier).
  - Si le haut de pile est une **flèche** , on dépile et on note cette flèche dans le ruban du bas (dans le sens d'écriture).

Voici le tableau pour $n = 3$:

Rappel des règles:

- 0 donne ←
- 1 donne 0 ↑
- 2 donne 1 0 →
- 3 donne 2 1 ↓

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			0	←										
		1	↑	↑	↑									
		0	0	0	0	**0**	←							
	2	→	→	→	→	→	→	→		**0**	←			
	1	1	1	1	1	1	1	1	**1**	↑	↑	↑		
3	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	fin
←	↑	←	→	←	↑	↓								

Trouvez la trajectoire du pingouin pour $n = 4$.

Exercice 3:

Segmentation fault

Vous venez de terminer un exercice de programmation et votre code ne fonctionne pas, et l'erreur ne vous éclaire pas davantage : "Segmentation fault". Trouvez l'origine du bug.

Listing 2: Faulty realloc

```
void realloc_dynamic_array(void *pointer, size_t new_size) {
    pointer = realloc(pointer, new_size);
    // Do some stuffs here
}

// The seg fault occurs a few lines after the function call
```

Aide pour la syntaxe : ***pointer*** est une adresse, ***new_size*** est un entier positif. La fonction **realloc** prend l'adresse d'une zone mémoire allouée dynamiquement, change sa taille, copie potentiellement les données ailleurs si besoin et renvoie l'adresse de la nouvelle zone mémoire.

Exercice 4:

Solveur d'opérations

Dans cet exercice, on se propose de transformer une chaine de caractère du type "(((3+4)/((((2+4)-5)*4)/2))" en un résultat à virgule flottante. Une expression "EXPR" peut prendre quatre formes. En reprenant la formulation des pingouins :

- EXPR donne (num op num)
- EXPR donne (EXPR op num)
- EXPR donne (num op EXPR)

- EXPR donne (EXPR op EXPR)

Avec "op" pouvant être un des symboles +, -, * et / pour additionner, soustraire, multiplier et diviser. et "num" un chiffre entre 0 et 9. On vous fait remarquer que la première parenthèse fermante rencontrée ferme une expression du type (num op num). En réutilisant la méthode du tableau pour visualiser la pile d'exécution, essayez de trouver une méthode (en langage naturel) pour implémenter la transformation. Vous n'avez plus besoin du ruban, annotez chaque colonne dans le sens de lecture avec les caractères de la chaîne lue. Essayez avec ((3*4)/(4+3)).

2 Partie 2: TP

Exercice 5:

Implémentation d'une pile

Utilisez le squelette ci-dessous pour implémenter un objet Stack (pile). Vous implémenterez les **fonctions** *length, is_empty, push, pop, et lookup.* Vous n'utiliserez pas d'indices négatifs, ni **len** ou **not** sur *data_array.* Vous ne pouvez pas lire les valeurs dans "*data_array*", seulement les renvoyer ou en ajouter. Utilisez "raise Exception (message)" pour les cas d'erreur.

Listing 3: Stack

```
class Stack
    def __init__(self, max_size: int):
        self.data_array: list[any] = [None for i in range(max_size)]
        # todo

def length(stack: Stack):
    # todo

# etc.
```

Exercice 6:

Lorem ipsum

Un lorem ipsum est un texte arbitrairement long, ressemblant à du latin et **sans signification** ("A bove ante, ab asino retro, a stulto undique caveto" n'est apparemment donc pas un lorem ipsum) utilisé pour tester la mise en page en imprimerie (ou en LaTeX...). On se propose de compter le nombre d'occurrences d'une lettre donnée dans un lorem ipsum. Comparez les deux méthodes suivantes :

Listing 4: Text formating

```
def rec_count(idx: int, count: int, string: str, letter: str):
    if idx == len(string):
        return count
    if string[idx] == letter:
        count += 1
    return rec_count(idx + 1, count, string, letter)

def ite_count(string: str, letter: str):
    count = 0
    # (pythonic short form would be "for c in string")
    for idx in range(len(string)):
        if string[idx] == letter:
            count += 1
    return count
```

Exercice 7:

Rocket Santa

Vous vous proposez d'écrire une fonction utilisant une pile pour former un palindrome à partir d'une chaine de caractères et d'un caractère optionnel qui se retrouverait au milieu du palindrome. La chaîne constitue le début du palindrome, le caractère au milieu peut être une chaîne vide.

Exercice 8:

Evil Rocket Santa

Pour la beauté du sport, vous vous proposez maintenant d'écrire une fonction utilisant une pile afin vérifier qu'une chaine est un palindrome (avec 2 boucles parcourant len(mot) // 2 caractères maximum).

Exercice 9:

Recursive when required, iterative when possible?

Une technique classique pour identifier un palindrome est d'inverser la chaine de caractère et de la comparer avec l'original. Proposez une **autre** solution itérative. Comparer cette solution avec la technique d'inversion.

Exercice 10:

Parenthésages

Vous commencez à comprendre le fonctionnement d'une pile, on vous "propose" de créer une fonction pour vérifier qu'un parenthésage est correct (e.g. " ([ ] [{ }]) " est correct").

Astuce : il peut être pratique d'avoir un dictionnaire avec un mapping judicieusement choisi :

Listing 5: parenthesis maps

```
p_num = {
    '(': 1,
    ')': -1,
    '[': 2,
    ']': -2,
    '{': 3,
    '}': -3
}
```

Exercice 11:

So it begins

Reprenez l'exercice sur le solveur d'opérations et implémentez l'algorithme en Python. Pour simplifier, maintenez une pile de chaines de caractères et utilisez float() pour convertir certains éléments que vous récupérez de la pile ainsi que str() avant d'empiler.