Exercice 1

Labyrinthe

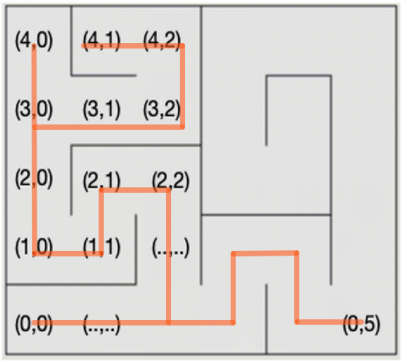

FIGURE 1 – sortir d'un labyrinthe

En adoptant la règle suivante :

- s'il y a une seule direction possible, avancer (couloir)
- s'il y a plusieurs directions possibles : choisir l'une des directions non parcourue (au hasard)
- impasse : revenir en arrière et choisir l'une des directions non parcourues.

> La tortue prend à gauche à la premiere bifurcation [1]. Arrive à une impasse [2]. Puis revient en arrière jusqu'à cette bifurcation [3]. Elle poursuit alors tout droit, ...etc...

On utilise l'implémentation fonctionnelle vue en cours.

1. Pour arriver au [1], on empile dans P les coordonnées (4,0) et (3,0). Donner les instructions correspondantes.
2. Représenter la pile P aux moments : [1], et [2].
3. La tortue est sur [2]. Combien de fois faudra-t-il depiler pour revenir à (3,0) ?

Pour réaliser les 2 prochains mouvements, il faut executer les instructions :

```
>>> empile(P,(2,0))
>>> empile(P,(1,0))
```

4. Quel est alors l'état de la pile P ?

Exercice 2

Connaissance du cours

1. Citer les opérations qui font partie de l'interface d'une *pile*.
2. Vrai ou Faux ? Pour ajouter un élément à une pile, il est aussi facile de l'ajouter au début qu'à la fin de la pile.
3. Vrai ou Faux ? L'opération *dépiler* (ou `pop`) s'execute en un temps qui est proportionnel au nombre de valeurs stockées dans la pile ?
4. Vrai ou Faux ? Pour accéder à un élément d'une pile, de nom `ma_pile` il suffit de connaitre son indice `i` et de faire : `ma_pile[i]`.
5. Vrai ou Faux ? Pour accéder au dernier élément d'une pile, l'opération s'effectue en un temps constant, indépendant de la taille de la pile.
6. Vrai ou Faux ? L'opération sur une liste `liste.pop(0)` permet de supprimer et retourner le premier élément d'une liste
7. Vrai ou Faux ? Cette opération se fait en un TEMPS CONSTANT.

Exercice 3

Pile d'instructions

Considérons l'exemple suivant :

```
1 def h(x):
2   return x+1
3
4 def g(x):
5   return h(x)+2
6
7 def f(x)
8   return g(x)+1
```

1. Que se passe-t-il lors de l'appel `f(5)` ? Représenter la pile d'instructions correspondante.
2. Calculer le résultat pour `f(5)`

Exercice 4

Expression correctement parenthesée

4.1 Parenthèses simples

a) Écrivez une fonction `parentheses_equilibrees(expression)` qui vérifie si une expression contient des parenthèses bien équilibrées.

Exemples :

- `"(()())"` ⟶ True

- `"(()"` → False
- `"())()"` → False
- `"(a + (b * c))"` → True

Algorithme à utiliser : - Parcourir l'expression caractère par caractère - Si on rencontre `(`, l'empiler - Si on rencontre `)`, dépiler (si la pile est vide, c'est déséquilibré) - À la fin, la pile doit être vide

```
def parentheses_equilibrees(expression):
    pile = []
    # À compléter
    pass
```

b) Tester votre fonction avec l'expression mathématique g :

```
>>> g = ((1+2)*3) + 10*(3-1))
>>> parentheses_equilibrees(g)
False
```

4.2 Parenthèses multiples

On cherche maintenant à vérifier l'équilibrage de plusieurs types de délimiteurs : `()`, `[]`, `{}`.

Exemples :

- `"{[()]}"` → True
- `"{[(])}"` → False (mauvais ordre de fermeture)
- `"[({})]"` → True

On utilisera un tableau associatif `dicoS` comprenant les couples de symboles associés, ainsi qu'une pile `p`.

1. Ecrire le contenu du dictionnaire `dicoS`. Les clés seront les symboles ouvrants [, { et (et les valeurs, les caractères fermants], },)
2. Ecrire une boucle bornée qui parcourt les caractères de la chaine `g`
3. Puis le contenu de cette boucle : si le caractère est une parenthèse *ouvrante*, empiler dans `p`. Si le caractère est fermant, dépiler `p` à condition que ce caractère corresponde à celui qui est au sommet de la pile `p`.
4. A quelle condition sur `p` peut-on déduire que l'expression est correctement parenthésée ?
5. Ecrire une fonction qui prend en argument une chaine de caractères représentant l'expression mathématique, et qui renvoie le booléen True si celle-ci est correctement parenthésée.
6. Ajouter les instructions à cette fonction pour que celle-ci lève une exception dans le cas où l'expression n'est PAS correctement parenthésée.

Exercice 5

Notation polonaise inversée (NPI)

La notation polonaise inversée place les opérateurs après les opérandes. Par exemple :

- `3 4 +` signifie `3 + 4 = 7`
- `5 2 * 3 +` signifie `(5 * 2) + 3 = 13`

Écrivez une fonction `evaluer_npi(expression)` qui évalue une expression en NPI.

Algorithme :

- Parcourir l'expression mot par mot
- Si c'est un nombre, l'empiler
- Si c'est un opérateur (+, -, *, /), dépiler deux valeurs, calculer, et empiler le résultat
- Le résultat final est au sommet de la pile

Exemples :

```
evaluer_npi("3 4 +")          # 7
evaluer_npi("15 7 1 1 + - /")  # 3  car 15 / (7 - (1 + 1))
evaluer_npi("5 2 * 3 +")      # 13
```

```
def evaluer_npi(expression):
    pile = []
    tokens = expression.split()
    # À compléter
    pass
```