TD - 02

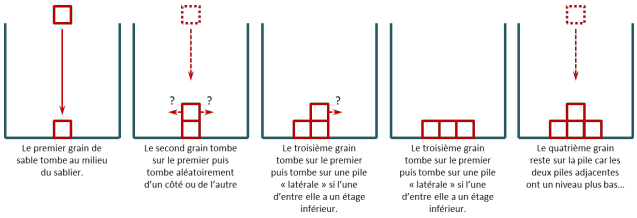
Savoirs et compétences :

- ❒ *Alg – C16 : Piles - Algorithmes de manipulation : fonctions de gestion des piles avec les méthodes des listes.*

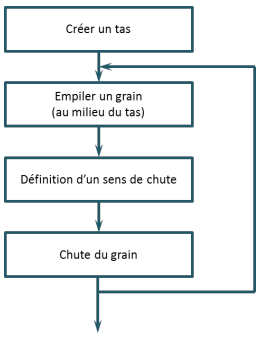
Exercice 1 - Le marchand de sable

On se propose de modéliser la constitution d'un tas de sable ainsi que l'écoulement des grains dans un sablier. Afin de simplifier le problème, on se restreindra à travailler en 2 dimensions. Le tas sera modélisé par une pile de grains de sable.

Dans le cas du sablier, les grains tombent toujours sur la même pile. Le processus de constitution de la pile est le suivant :



Un algorithme très succinct présente le déroulement de la chute d'un grain de sable.



Création des objets

Gestion des piles de sables

Une pile de sable est modélisée par... une pile ! Cette dernière est implémentée sous forme d'une liste.

Question 1 *Donner l'implémentation des fonctions élémentaires permettant de créer une pile dans Python à savoir les fonctions* `creer_pile`, `empiler`, `depiler`, `est_vide`. ***Pour cette question on s'autorise l'utilisation des méthodes sur les listes.***

Question 2 *Donner l'implémentation de la fonction* `taille_pile`, *permettant de connaître la taille d'une pile.* ***Pour cette question seules les fonctions définies ci-dessus sont acceptées.*** *Donner la complexité algorithmique de la fonction implémentée.*

Question 3 *Redéfinir la fonction* `empiler` *en la nommant* `empilerSable` *pour que le seul élément empilable soit la chaîne de caractères "*". Ainsi, une pile de sable sera constituée d'une pile d'étoiles.*

Gestion du tas de sable

Un tas de sable va être modélisé par une **liste de piles** de grains de sable. Pour cette liste, les méthodes de liste peuvent être utilisées.

Question 4 *Implémenter la fonction* `creer_tas` *d'argument* `n` *permettant de créer un tas de* `n` *piles de sables.*

Question 5 *Donner les instructions permettant de dépiler un grain de sable sur la pile i et d'empiler un grain de sable sur la pile $i-1$ du tas à n piles. Tester les instructions pour la pile $i = 2$ du* `tasSable=[[],["*"],["*","*","*"],["*"],["*","*"]]`.

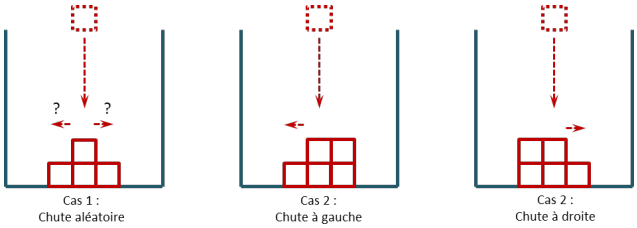
Écoulement

On va maintenant implémenter les fonctions qui vont permettre de régir l'écoulement d'un grain de sable. On suppose que les grains tombent toujours sur la même pile.

On s'intéresse d'abord au sens d'écoulement d'un grain de sable. Pour cela, on définit une variable `sens`

qui vaut 0 lorsque le grain doit s'écouler vers la gauche et qui vaut 1 lorsque le grain doit s'écouler vers la droite.

On identifie les 3 cas suivants pour déterminer le sens de chute d'un grain :



On compare la taille des piles avant que le grain de sable soit tombé.

Question 6 *Exprimer la condition booléenne pour laquelle un grain de sable chute à gauche.*

Question 7 *Exprimer la condition booléenne pour laquelle un grain de sable chute aléatoirement à gauche ou à droite.*

Question 8 *En réalisant un schéma, donner un cas de figure pour lequel il n'y a pas d'écoulement de grain. Traduire la condition booléenne correspondante.*

Question 9 *Exprimer la condition booléenne permettant de savoir si un grain qui tomberait sur la pile n doit s'écouler sur la gauche. On tiendra compte du cas où le grain est sur le bord du sablier.*

Question 10 *Implémenter la fonction* `sens` *permettant de déterminer le sens de la chute du grain : droite, gauche ou si le grain tombe sur sa pile. Cette fonction prendra comme arguments tas(liste de piles) et indice(int) l'indice de la pile sur laquelle le grain sera laché.*

On appelle `chute` la fonction permettant de régir la chute du grain. Les spécifications de la fonction sont les suivantes :

▪ **Python**
```
def chute(tas,indice):
    """
    Gestion d'une chute de grain de sable.
    Entrées :
     * tas(liste de piles) : tas de sable
     * indice(int) : pile sur laquelle le dernier
                grain de sable va tomber
     * sens(int) : 0 chute àgauche, 1 chute à
droite
                None si le grain reste sur la pile
    Sortie :
     * le tas est modifié mais n'est pas retourné.
    """
```

Question 11 *Implémenter la fonction* `chute` *permettant de gérer la chute d'un grain de sable.* ***Cette fonction devra être récursive****. Tester la fonction* `chute` *à partir d'un tas de sable de 7 piles sur lequel vous ferez chuter 12 grains de sable sur la pile d'indice 3.*

Affichage du tas de sable

On donne le tas suivant :

▪ **Python**
```
>> print(tas)
    [[],['*'],['*','*'],['*','*','*'],
                ['*','*'],['*'],[]]
```

On souhaite l'afficher sous la forme suivante :

▪ **Python**
```
-------
___*___
__***__
_*****_
```

Question 12 *Implémenter la fonction* `affichage` *permettant d'afficher un tas sous la forme définie ci-dessus. Tester l'affichage du tas créé à la question précédente.*

Exercice 2 – Notation polonaise inversée

La notation polonaise inversée permet de réaliser des opérations arithmétiques sans utiliser de parenthèses. On parle aussi d'expressions **postfixées**. Dans cette notation, l'opérateur (+, -, *, /) suit toujours l'opérande (nombres). Ici, l'opération suit le deuxième opérande.

On doit évaluer une expression composée d'opérateurs et d'opérandes. L'expression est lue de gauche à droite en suivant les règles suivantes :

- si on lit une opérande, on l'empile ;
- si on lit un opérateur, on dépile deux opérandes, on réalise le calcul et on empile le résultat.

▪ **Exemple** :
Pour calculer 7×2, on évalue l'expression $7\,2\,*$.
Pour calculer $((1+2)\times 4)+3$, on évalue l'expression $1\,2\,+\,4\,*\,3\,+$.
Pour calculer $((1-2)\times(4+5))$, on évalue l'expression $1\,2\,-\,4\,5\,+\,*$.
Pour calculer $((3+(5\times(7+2)))+(4+(8\times 9))$, on évalue l'expression $3\,5\,7\,2\,+\,\times\,+\,4\,8\,9\,\times\,+\,+$. ▪

R Le principal avantage de cette méthode est la suppression totale des parenthèses. En contrepartie, elle nécessite une petite gymnastique intellectuelle pour les néophytes que nous sommes.

Cette notation est utilisée dans certaines calculatrices HP et dans certains utilitaires (programme calc d'Emacs, description des bibliographies dans LaTeX *etc.*).

Dans cet exercice, la pile sera implémentée sous forme d'une liste d'opérande et d'opérateurs.

On utilisera les fonctions définies à l'exercice précédent.

Question 1 *Créer la fonction* `est_nombre` *permettant de savoir si une variable est un nombre ou non.*

Question 2 *Créer la fonction* `est_operation` *permettant de savoir si une variable est une opération ou non.*

Question 3 *L'expression postfixée se lisant de gauche à droite et les outils permettant de traiter la pile traitant cette dernière de droite à gauche, réaliser la fonction* `inversion` *permettant d'inverser les éléments de la pile.*

Question 4 *Créer la fonction* `evaluer` *permettant d'évaluer une expression postfixée. Les spécifications sont les suivantes :*

▪ Python
```
def evaluer(exp):
    """
    Évaluer le résultat d'une opération post-fixée.
    Entrée :
     * ex(lst) : liste d'opérateurs et d'opérandes
    Sortie :
     * res(flt) : résultat du calcul de l'expression.
    """
```

Question 5 *Transcrire l'expression suivante en NPI puis l'évaluer :* $(1+2)\times(3\times 10)+4$.

Question 6 *Analyse d'une fonction récursive... à partir d'un arbre binaire.*