T.P. d'informatique n° 1

Histoires de piles et mots de Dyck.

On appelle **mot de Dyck** tout mot construit à l'aide des deux lettres **E** et **D** vérifiant les deux propriétés suivantes :

• le mot contient autant de « **E** » que de « **D** »
• dans tout préfixe du mot, le nombre d'occurrences de « **E** » est supérieur ou égal au nombre d'occurrences de « **D** ».

Comme pour un langage classique, on appelle ***alphabet*** *l'ensemble des lettres utilisées. Ici, il s'agit donc de l'alphabet à deux lettres* $\mathcal{A} = \{E, D\}$.

Quest. 1. Les mots « **ED** », « **EDED** », « **EEDE** », « **EDDE** », « **EEDD** », « **EDEEDD** », « **EEEDDD** », « **EDEEDDEDD** », « **EDEDDEED** » et « **EEEDEDDEDDEEDDED** » sont-ils de Dyck ?

Quest. 2. Une suite d'opérations d'empilement et de dépilement sur une pile peut aisément s'écrire par un mot sur l'alphabet **{E,D}** où "**E** " indique un empilement et " **D** " un dépilement.
Par exemple si **p** est une pile d'entiers, la suite d'opérations suivante : « empiler(1), dépiler, empiler(3), empiler(5) » se traduira simplement par le mot « **EDEE** ».
Il est évident que le mot « **EDEE** » contient la suite des opérations (empilement/dépilement) sans souvenir des valeurs empilées...
La suite des opérations d'empilement et dépilement est dite **histoire de la pile** et le mot défini comme précédemment est dit **associé à l'histoire de la pile**.

Montrer qu'un mot sur l'alphabet **{E,D}** est associé à une histoire de pile commençant et se terminant par la pile vide si et seulement si c'est un mot de Dyck.

Quest. 3. Écrire une fonction Python **deDyck** reçevant un mot et retournant un booléen testant si le mot passé en argument est un mot de Dyck.

Quest. 4. Montrer que la concaténation de deux mots de Dyck est un mot de Dyck.

Quest. 5. Montrer que si un mot de Dyck **mot** est la concaténation du préfixe **mot1** et du suffixe **mot2** et si **mot1** est un mot de Dyck alors **mot2** est aussi un mot de Dyck.

Quest. 6. Montrer qu'un mot de Dyck est de longueur paire, commence par la lettre **E** et finit par la lettre **D**.

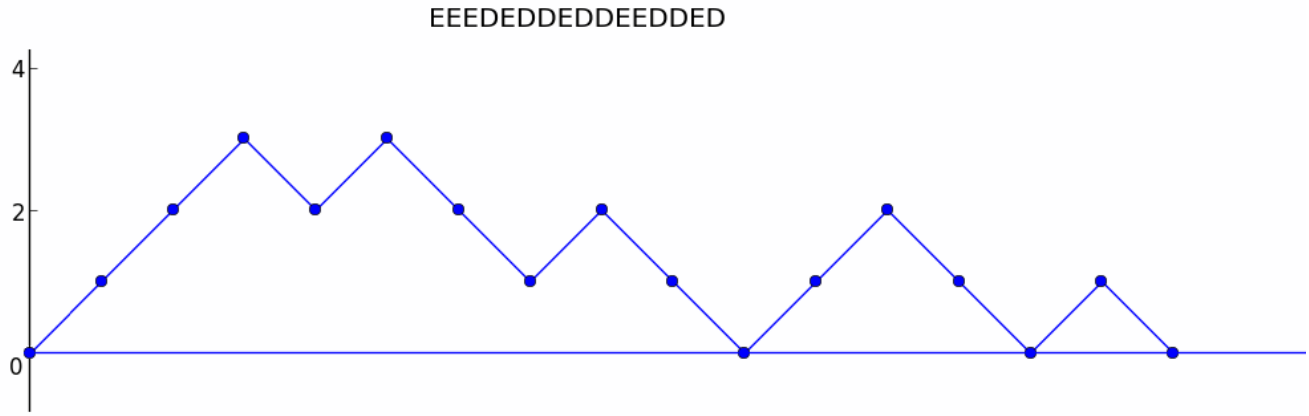
Quest. 7. Mots de Dyck de longueur donnée

1. Le mot vide est le seul mot de Dyck de longueur 0. Il n'y a qu'un seul mot de Dyck de longueur 2 (**ED**) et deux mots de Dyck de longueur 4 (**EDED** et **EEDD**).

   Soit n un entier naturel. On note c_n le nombre de mots de Dyck de longueur $2\,n$. Montrer alors la formule :$c_n = \sum_{k=0}^{n-1} c_k\, c_{n-1-k}$.
   On pourra montrer que les mots de Dyck de longueur $2\,n$ *sont les mots de la forme* ***Emot3Dmot2*** *avec* ***mot3*** *et* ***mot2*** *mots de Dyck indépendants de longueurs respectives* $2\,k$ *et* $2\,(n-k-1)$ *où* k *décrit* $[\![0, n-1]\!]$.

2. Écrire une fonction Python **nbMDD** recevant un entier **N** et retournant le nombre de mots de Dyck de longueur **2 N**.
3. Écrire une fonction Python **listeMDD** recevant un entier **N** et retournant la liste de tous les mots de Dyck de longueur **2 N**.

Quest. 8. [Uniquement pour les plus rapides ou à voir après le TP]
On peut représenter graphiquement un mot de Dyck par une ligne brisée de pente $+1$ en cas d'empilement et de pente -1 en cas de dépilement ; après avoir chargé le module de la tortue par l'instruction : **from turtle import ***, puis avoir pris connaissance sur internet des fonctions **clear**, **up**, **down**, **goto** et **color**, écrire une suite d'instructions permettant de tracer la ligne brisée associée à un mot de Dyck reçu en argument.
On pourra aussi, si on préfère, utiliser le module **matplotlib**.



Autres Exercices

1. **La notation polonaise inverse**
   La **notation polonaise inverse** (**NPI**) permet d'écrire de façon non ambiguë les formules arithmétiques sans utiliser de parenthèses.
   Par exemple l'expression « $1+6\times 5$ » s'écrit « 1 6 5 $\times$ + », l'expression « $(1+6)\times 5$ » s'écrit « 1 6 + 5 $\times$ » et enfin l'expression « $(2\times(4+6))/5$ » s'écrit « 2 4 6 + $\times$ 5 / ».
   (a) Ecrire en NPI les expressions suivantes :
   $$(1+6)\times(5+4\times 3),\ ((1+2)\times 3+4)\times 5,\ (1+6)-(5+9\times 4) \text{ et } 9\times 5/(4+3).$$
   (b) Evaluer les expressions suivantes écrites en NPI :
   $$5\ 10+,\ 2\ 10\ 4+-,\ 10\ 4+2-\ \text{et}\ 4\ 3+3\ 2+\times.$$
   (c) On cherche maintenant à écrire une fonction permettant d'évaluer une expression donnée en NPI. Pour cela, on suppose que l'expression est donnée sous la forme d'une liste. Par exemple, on représentera l'expression « 10 4 + 2− » par $[10, 4, '+', 2, '-']$. On effectue alors les actions suivantes :
   - ✎ On parcourt la liste
   - ✎ Si on rencontre un nombre, on le place dans une pile (vide au début)
   - ✎ Si on recontre un opérateur (+, -, *, /), il faut alors dépiler deux éléments de la pile, effectuer l'opération et empiler le résultat.
   - ✎ Une fois la liste parcourue, la pile ne contient plus qu'un élément qui est la valeur souhaitée.
     On admettra que les expressions entrées en NPI sont correctes.

   i. Tester cette procédure sur des exemples.
   ii. Construire une liste de deux listes **op**
   - ✎ **op[0]** valant ['+','-', '*', '/']
   - ✎ **op[1]** valant la liste des fonctions correspondantes $[add, sou, fois, sur]$ où par exemple add peut-être définie par

```
def add(x,y):
    return (x + y)
```

iii. Ecrire une fonction **evalNPI** prenant en argument une liste représentant une expression écrite en NPI et renvoyant la valeur de cette expression.

2. **Jeu de cartes**
Importer le module `random` qui contient des commandes générant des nombres aléatoires.
Par exemple l'appel `randint(a,b)` où a et b sont des entiers, renvoie un entier choisi aléatoirement dans $[\![a, b]\!]$

(a) Créer une pile représentant une couleur de carte : $p = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]$

(b) Écrire une fonction `couper(pile)` qui coupe une pile en deux piles. La coupure est aléatoire mais les deux piles devront être non vides (lorsque cela est possible). De la pile de départ ne resteront que les premiers termes (le nombre de termes restant est le nombre aléatoire choisi). La seconde pile présente les éléments restants dans l'ordre inverse. Par exemple : In [1] : couper([1,2,11,4,13,6,7])
Out[1] : ([1,2,11] , [7,6,13,4])

(c) Écrire une fonction `melange` qui prend en entrée un couple de deux piles p_1 et p_2, et qui renvoie une pile p formée du mélange suivant :

- ✎ tant qu'aucune pile n'est vide, on dépile alátoirement p_1 ou p_2 et on empile sur p
- ✎ si l'une des deux piles d'entrée est vide, on dépile les éléments restants de l'autre sur p
- ✎ On inverse p

In [1] : melange([1,2,3,4],[5,6,7])
Out[1] : [1,5,2,3,6,4,7]

(d) Tour de magie de Gilbreath
Créer une pile formée de n répétitions de k cartes.
Couper puis mélanger cette pile selon les procédures précédentes. Remarquer que l'on retrouve dans la pile résultat n répétitions successives des k cartes, éventuellement mélangées au sein d'une répétition.
Exemple :
In [1] : Gilbreath([1,2,3,1,2,3,1,2,3,1,2,3])
Out[1] : [3,2,1,3,1,2,1,3,2,3,2,1]

(e) **Bataille**

i. Créer un jeu complet de 52 cartes (4 ensembles de 13 cartes). Le mélanger plusieurs fois.
Deux joueurs découvrent alternativement le sommet du jeu. Le joueur dont la carte possède la valeur la plus forte marque un point.

ii. Implémenter un jeu de bataille prenant un jeu de cartes mélangées en entrée et retournant les scores des deux joueurs et proclamant le joueur gagnant

1 Correction des exercices

Autres exercices

1. Ecrire, en anticipant les différents cas qui peuvent provoquer des erreurs, les fonctions suivantes :
   (a) `taille(p)` : déjà fait. Si n est la longueur de la pile, on effectue $2n$ dépilements, $2n$ empilements, n additions donc on a une complexité en temps en $O(n)$. LA complexité en mémoire est $2n$ car on gère deux piles qui ont potentiellement cette taille.
   Dans la version des piles à capacité finie, la complexité en temps est $O(1)$ et en mémoire $O(n)$ où n n'est pas la taille de la pile mais le maximum de cette taille.
   (b) `sommet(p)` : déjà écrite.
   (c)
```
def vide_pile(p):
  while not est_vide(p):
    depiler(p)
```
   (d)
```
def inverser(p):
  s=creer_pile()
  while not est_vide(p):
    empiler(s,depiler(p))
  return s
```
   (e)
```
def intervertir_sommet(p):
  if not est_vide(p):
    s=depiler(p)
    if not est_vide(p):
      t=depiler(p)
      empiler(p,s)
      empiler(p,t)
    else:
      empiler(p,s)
```
   (f)
```
def couper(p,n):
  i=0
  s=creer_pile()
  while i<n:
    if not est_vide(p):
      empiler(s,depiler(p))
      i+=1
    else:
      while not est_vide(s):
        empiler(p,depiler(s))#on
            ne modifie pas p dans
             ce cas
      return("n est trop grand")
  t=creer_pile()
  while not est_vide(s):
    empiler(t,depiler(s))
  return t
```
   (g)
```
def lit_element(p,n):
  i, s = 0, creer_pile()
  while i<n:    #l'element 0 est
     le sommet
    if not est_vide(p):
      empiler(s,depiler(p))
      i+=1
    else:
      while not est_vide(s):
        empiler(p,depiler(s))
      return("n est trop grand")
  if not est_vide(p):
    r=depiler(p)
    empiler(p,r)
  else:
    while not est_vide(s):
      empiler(p,depiler(s))
    return("n est trop grand")
  while not est_vide(s):
    empiler(p,depiler(s))
  return(r)
```

2. Implémenter un test de bon parenthésage `parenthese(chaine)`

```
def testParenthese(chaine):
  p = creer_pile()
  erreur = False
  for i in range(len(chaine)):
    if chaine[i] == '(' :
      empiler(p,1)
    elif chaine[i] == ')' and not(est_vide(p)) :
      depiler(p)
    elif chaine[i] == ')' and est_vide(p) :
      erreur = True
    if not(est_vide(p)) or erreur :
      print('syntaxe incorrecte')
    else :
      print('syntaxe correcte')
```

3. **La notation polonaise inverse**

```
def NPI(donnee):
 pile = creer_pile()
 for x in donnee :
   if x not in '+,-,*,/':
     empiler(pile,x)
   else:
     operation = x
     b = depiler(pile)
     a = depiler(pile)
     if operation == '+':
       empiler(pile, a+b)
     elif operation == '*':
       empiler(pile, a*b)
     elif operation == '/':
       empiler(pile, a/b)
 return(depiler(pile))
```

4. **Jeu de cartes**

Importer le module `random` qui contient des commandes générant des nombres aléatoires.
Par exemple l'appel `randint(a,b)` où a et b sont des entiers, renvoie un entier choisi aléatoirement dans $[\![a, b]\!]$

(a) Créer une pile représentant une couleur de carte : $p = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]$

(b) Écrire une fonction `couper(pile)` qui coupe une pile en deux piles. La coupure est aléatoire mais les deux piles devront être non vides (lorsque cela est possible). De la pile de départ ne resteront que les premiers termes (le nombre de termes restant est le nombre aléatoire choisi). La seconde pile présente les éléments restants dans l'ordre inverse. Par exemple : In [1] : couper([1,2,11,4,13,6,7])
Out[1] : ([1,2,11] , [7,6,13,4])

(c) Écrire une fonction `melange` qui prend en entrée un couple de deux piles p_1 et p_2, et qui renvoie une pile p formée du mélange suivant :

- ✎ tant qu'aucune pile n'est vide, on dépile alátoirement p_1 ou p_2 et on empile sur p
- ✎ si l'une des deux piles d'entrée est vide, on dépile les éléments restants de l'autre sur p
- ✎ On inverse p

In [1] : melange([1,2,3,4],[5,6,7])
Out[1] : [1,5,2,3,6,4,7]

(d) Tour de magie de Gilbreath
Créer une pile formée de n répétitions de k cartes.
Couper puis mélanger cette pile selon les procédures précédentes. Remarquer que l'on retrouve dans la pile résultat n répétitions successives des k cartes, éventuellement mélangées au sein d'une répétition.
Exemple :
In [1] : Gilbreath([1,2,3,1,2,3,1,2,3,1,2,3])
Out[1] : [3,2,1,3,1,2,1,3,2,3,2,1]

(e) **Bataille**

i. Créer un jeu complet de 52 cartes (4 ensembles de 13 cartes). Le mélanger plusieurs fois.
Deux joueurs découvrent alternativement le sommet du jeu. Le joueur dont la carte possède la valeur la plus forte marque un point.

ii. Implémenter un jeu de bataille prenant un jeu de cartes mélangées en entrée et retournant les scores des deux joueurs et proclamant le joueur gagnant

Mercredi 16 septembre 2015. Spé MP.

T.P. d'informatique n° 1

Histoires de piles et mots de Dyck.

I Mots de Dyck

On appelle **mot de Dyck** tout mot construit à l'aide des deux lettres **E** et **D** vérifiant les deux propriétés suivantes :

- le mot contient autant de « **E** » que de « **D** »
- dans tout préfixe du mot, le nombre d'occurrences de « **E** » est supérieur ou égal au nombre d'occurrences de « **D** ».

Comme pour un langage classique, on appelle **alphabet** *l'ensemble des lettres utilisées. Ici, il s'agit donc de l'alphabet à deux lettres* $\mathcal{A} = \{E, D\}$.

Question 1 : Les mots « **ED** », « **EDED** », « **EEDE** », « **EDDE** », « **EEDD** », « **EDEEDD** », « **EEEDDD** », « **EDEEDDEDD** », « **EDEDDEED** » et « **EEEDEDDEDDEEDDED** » sont-ils de Dyck ?

Solution : Les mots « **ED** » et « **EDED** » sont de Dyck ;
« **EEDE** » n'est pas de Dyck car il contient 3 **E** et 1 **D** ;
« **EDDE** » n'est pas de Dyck car le préfixe **EDD** ne vérifie pas la seconde propriété ;
« **EEDD** », « **EDEEDD** », « **EEEDDD** » sont de Dyck ;
« **EDEEDDEDD** » n'est pas de Dyck car il contient 4 **E** Et 5 **D** ;
« **EDEDDEED** » n'est pas de Dyck car le préfixe **EDEDD** ne vérifie pas la seconde propriété ;
« **EEEDEDDEDDEEDDED** » est de Dyck.

Question 2 : Montrer qu'un mot sur l'alphabet **{E,D}** est associé à une histoire de pile commençant et se terminant par la pile vide si et seulement si c'est un mot de Dyck.

Solution : Si la pile est vide au début et à la fin de l'histoire, il y a eu autant d'empilements que de dépilements, donc autant de « **E** » que de « **D** » dans le mot associé à l'histoire de la pile. Comme on ne peut pas dépiler sur une pile vide, à tout moment de l'histoire de la pile, le nombre d'empilements est supérieur ou égal au nombre de dépilements. Donc dans tout préfixe d'un mot associé à une histoire de pile, le nombre d'occurrences de « **E** » est supérieur au nombre d'occurrences de « **D** ».
Réciproquement, pour un mot de Dyck donné et partant d'une pile vide, comme dans tout préfixe du mot le nombre d'occurrences de « **E** » est supérieur au nombre d'occurrences de « **D** », à tout moment de l'histoire de la pile, le nombre d'empilements est supérieur ou égal au nombre de dépilements et donc on ne se retrouve jamais à devoir dépiler sur une pile vide (seule opération impossible, il est en effet toujours possible d'empiler...) et donc l'histoire de la pile est possible. Comme, il y a autant de « **E** » que de « **D** » dans le mot, il y a autant d'empilements que de

dépilements et donc, la pile (vide au début) est bien vide à la fin de l'histoire.

Question 3 : Écrire une fonction Python **deDyck** reçevant un mot et retournant un booléen testant si le mot passé en argument est un mot de Dyck.
Solution : En se basant sur l'histoire de la pile, on peut écrire la fonction **deDyck** suivante :

```
def deDyck(mot) :
    p=[]
    for l in mot :
        if l =='E' :
            p.append(1)
        elif l=='D' :
            if len(p)>0 :
                p.pop()
            else :
                return False
        else :
            return False
    if len(p)==0 :
        return True
    else :
        return False
```

Mais, en pratique, nul besoin de pile pour écrire la fonction **deDyck** : il suffit d'utiliser un compteur **c** donnant au cours de l'histoire de la pile la hauteur de la pile.

```
def deDyck2(mot) :
    c=0
    for l in mot :
        if l =='E' :
            c+=1
        elif l=='D' :
            if c>0 :
                c-=1
            else :
                return False
        else :
            return False
    if c==0 :
        return True
    else :
        return False
```

Question 4 : Montrer que la concaténation de deux mots de Dyck est un mot de Dyck.

Solution : Supposons que **mot** est la concaténation des deux mots de Dyck **mot1** et **mot2**.

On note **nbE** et **nbD** les fonctions donnant le nombre d'occurences respectives de **E** et de **D** d'un mot donné.
mot1 et **mot2** sont des mots de Dyck, donc :
nbE(mot1)=**nbD(mot1)** et pour tout préfixe **p1** de **mot1**, **nbE(p1)** $\geq$ **nbD(p1)**
nbE(mot2)=**nbD(mot2)** et pour tout préfixe **p2** de **mot2**, **nbE(p2)** $\geq$ **nbD(p2)** ;

alors, **nbE(mot)**=**nbE(mot1)**+**nbE(mot2)**=**nbD(mot1)**+**nbD(mot2)**=**nbD(mot)**
et pour tout préfixe **p** de **mot**,
si **p** est de longueur inférieure à celle de **mot1**, alors **p** est un préfixe de **mot1** et alors **nbE(p)** $\geq$ **nbD(p)**,
sinon **p** est la concaténation de **mot1** et **p2** avec **p2** préfixe de **mot2** et donc :
nbE(p)=**nbE(mot1)**+**nbE(p2)** $\geq$ **nbD(mot1)**+**nbD(p2)**=**nbD(p)** ;

ainsi, **mot** est un mot de Dyck, car
nbE(mot)=**nbD(mot)** et pour tout préfixe **p** de **mot**, **nbE(p)** $\geq$ **nbD(p)**.

Question 5 : Montrer que si un mot de Dyck **mot** est la concaténation du préfixe **mot1** et du suffixe **mot2** et si **mot1** est un mot de Dyck alors **mot2** est aussi un mot de Dyck.

Solution : **mot** est la concaténation du préfixe **mot1** et du suffixe **mot2**.
Or, **mot** est un mot de Dyck, donc
nbE(mot)=**nbD(mot)** et pour tout préfixe **p** de **mot**, **nbE(p)** $\geq$ **nbD(p)** ;
mot1 est un mot de Dyck donc **nbE(mot1)**=**nbD(mot1)** :
alors, **nbE(mot2)**=**nbE(mot)**-**nbE(mot1)**=**nbD(mot)**-**nbD(mot1)**=**nbD(mot2)**
et pour tout préfixe **p2** de **mot2**, si **p** est la concaténation de **mot1** et **p2**, alors **p** est un préfixe de **mot**, **mot1** est un mot de Dyck donc **nbE(mot1)**=**nbD(mot1)** :
alors, **nbE(p2)**=**nbE(p)**-**nbE(mot1)** $\geq$ **nbD(p)**-**nbD(mot1)**=**nbD(p2)**
donc **mot2** est aussi un mot de Dyck.

Question 6 : Montrer qu'un mot de Dyck est de longueur paire, commence par la lettre **E** et finit par la lettre **D**.

Solution : Soit **mot** un mot de Dyck,
sa longueur ℓ est ℓ=**nbE(mot)**+**nbD(mot)**=2**nbE(mot)** ;
soit **p** la première lettre de **mot**, c'est évidemment un préfixe de **mot**,
comme **nbE(p)** $\geq$ **nbD(p)** et **nbE(p)**+**nbD(p)**=1,
on a nécessairement **nbE(p)**=1 et **nbD(p)**=0 ;
soit maintenant **p** le préfixe de **mot** le longueur $\ell-1$ (où ℓ est la longueur de **mot**) et **q** la dernière lettre de **mot** (**mot** est la concaténation de **p** et **q**),
comme **nbE(p)** $\geq$ **nbD(p)**, **nbE(q)** $\leq$ **nbD(q)** et **nbE(q)**+**nbD(q)**=1,
on a nécessairement **nbE(q)**=0 et **nbD(q)**=1.

Question 7 : Mots de Dyck de longueur donnée

7.a. Le mot vide est le seul mot de Dyck de longueur 0. Il n'y a qu'un seul mot de Dyck de longueur 2 (**ED**) et deux mots de Dyck de longueur 4 (**EDED** et **EEDD**).

Soit n un entier naturel. On note c_n le nombre de mots de Dyck de longueur $2\,n$. Montrer alors la formule :

$$\boxed{c_n = \sum_{k=0}^{n-1} c_k\, c_{n-1-k}\ .}$$

Solution : D'après la remarque du début de la question, $c_0 = c_1 = 1$ et $c_2 = 2$.

Soit un mot de Dyck **mot** de longueur $2\,n$ avec $n \geq 1$: considérons le premier (par ordre de longueur croissante) préfixe **mot1** qui est un mot de Dyck non vide (un tel préfixe existe puisque au pire **mot** est un préfixe de lui même).

Comme **mot1** est un mot de Dyck, c'est la concaténation de la lettre **E** d'un mot **mot3** et de la lettre **D** : montrons alors que **mot3** est lui-même un mot de Dyck ;

mot1 est un mot de Dyck donc **nbE(mot1)**=**nbD(mot1)** et
nbE(mot3)=**nbE(mot1)**-1=**nbD(mot1)**-1=**nbD(mot3)**
si **mot3** n'est pas un mot de Dyck alors
il existe un préfixe **p3** de **mot3** tel que **nbE(p1)** < **nbD(p1)**
et alors **Ep1** est un préfixe de **mot1** et est un mot de Dyck non vide
(car **nbE(Ep1)** ≤ **nbD(Ep1)** et **nbE(Ep1)** ≥ **nbD(Ep1)** ⇒**nbE(Ep1)**=**nbD(Ep1)**)
ce qui est absurde puisque **mot1** est le premier.

Alors, **mot** est la concaténation du préfixe **mot1** et du suffixe **mot2** et, comme vu précédemment, **mot2** est aussi un mot de Dyck (c'est le mot vide si **mot1**=**mot**)
et on obtient donc **mot** =**Emot3Dmot2** **mot3** et **mot2** mots de Dyck de longueurs respectives $2\,k$ et $2\,(n-k-1)$ avec k entre 0 et $n-1$.

Réciproquement,
si **mot** =**Emot3Dmot2** avec **mot3** et **mot2** mots de Dyck alors **mot1** =**Emot3D** est le premier préfixe qui est un mot de Dyck non vide car, en effet, si **p** est un préfixe non vide de **Emot3** alors **p** est la concaténation de **E** et d'un préfixe **p'** de **mot3** vérifiant donc **nbE(p')** ≥ **nbD(p')** d'où **nbE(p)**=1+**nbE(p')** > **nbD(p')** =**nbD(p)** et donc **p** n'est pas un mot de Dyck.

Les mots de Dyck de longueur $2\,n$ sont donc les mots de la forme **Emot3Dmot2** avec **mot3** et **mot2** mots de Dyck indépendants de longueurs respectives $2\,k$ et $2\,(n-k-1)$ où k décrit $[\![0, n-1]\!]$:
ce qui donne ainsi la formule :

$$\boxed{c_n = \sum_{k=0}^{n-1} c_k\, c_{n-1-k}}$$

.

7.b. Écrire une fonction Python **nbMDD** recevant un entier **N** et retournant le nombre de mots de Dyck de longueur **2 N**.
Solution :

```
def nbMDD(N) :
    listC=[1]
    for n in range(1,N+1) :
        compt=0
        for k in range(n) :
            compt+=listC[k]*listC[n-k-1]
        listC+=[compt]
    return listC[N]
```

7.c. Écrire une fonction Python **listeMDD** recevant un entier **N** et retournant la liste de tous les mots de Dyck de longueur **2 N**.
Solution :

```
def listeMDD(N) :
    lMDD=[[""]]
    for n in range(1,N+1) :
        lis=[]
        for k in range(n) :
            for mot3 in lMDD[k] :
                for mot2 in lMDD[n-k-1] :
                    lis+=['E'+mot3+'D'+mot2]
        lMDD+=[lis]
    return lMDD[N]
```

Question 8 : [Uniquement pour les plus rapides ayant aussi fini la partie **II**]
On peut représenter graphiquement un mot de Dyck par une ligne brisée de pente $+1$ en cas d'empilement et de pente -1 en cas de dépilement ; après avoir chargé le module de la tortue par l'instruction : **from turtle import ***, puis avoir pris connaissance sur internet des fonctions **clear**, **up**, **down**, **goto** et **color**, écrire une suite d'instructions permettant de tracer la ligne brisée associée à un mot de Dyck reçu en argument.
On pourra aussi, si on préfère, utiliser le module **matplotlib**.

Solution : Solution **matplotlib** :

```
from matplotlib import pyplot as plt
def tracer(mot) :
    ymin,ymax=0,0
    c=0
    histoire=[0]
    for l in mot :
        if l =='E' :
            c+=1
        elif l=='D' :
            if c>0 :
                c-=1
            else :
                return None
        else :
            return None
        histoire+=[c]
        if c>ymax :
            ymax=c
        if c<ymin :
            ymin=c
    plt.plot([k for k in range(len(mot)+1)],histoire,'bo-')
    plt.axis('equal')
    plt.title(mot)
    plt.ylim(ymin-1,ymax+1)
    plt.axhline(y=0, xmin=0, xmax=len(mot))
    plt.show()
```