chProgDynamiqueNB

May 8, 2025

[ ]:
```
import pylab as pl
import time
```

1 2. Piles et files

1.1 Appli 2 : Notation polonaise inverse

[1]:
```
def RPN(l):
    pile=[]
    for el in l:
        if el not  in ('x','+'):
            pile.append(el)
        elif el=='+':
            pile.append(pile.pop()+pile.pop())
        else:
            pile.append(pile.pop()*pile.pop())
        print(pile)

RPN([2,3,4,'x','+'])
```

```
[2]
[2, 3]
[2, 3, 4]
[2, 12]
[14]
```

2 3. Dictionnaires

2.1 3.3. Fonction de hachage

Différents types peuvent être hachés

[2]:
```
hash(12345), hash("12345"), hash((1,2,3,4,5))
```

[2]:
```
(12345, -7313169373550564608, -5659871693760987716)
```

Mais pas les listes

[3]:
```
hash([1,2,3,4,5])
Traceback (most recent call last):
File"<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

```
  Cell In[3], line 2
    Traceback (most recent call last):
               ^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

La fonction de hachage est localement injective, c'est-à-dire qu'elle est très sensible aux petites variations des clefs.

[ ]:
```
hash("Dupont"),hash("Dupond"),hash("Dupont")-hash("Dupond")
```

[ ]:
```
(8826115988355940694, -550959240832759317, 9377075229188700011)
```

2.2 3.4 Les dictionnaires en pratique

2.2.1 3.4.1 Création d'un dictionnaire

[ ]:
```
prix = {"carottes": 1.49 ,"brocolis":2.49 ,"pommes":0.99}
dico_vide = {}
dico_vide2 = dict()

prix,dico_vide,dico_vide2
```

[ ]:
```
({'carottes': 1.49, 'brocolis': 2.49, 'pommes': 0.99}, {}, {})
```

[ ]:
```
prix["pommes"]
```

[ ]:
```
0.99
```

[ ]:
```
prix["avocats"] = 4.99
print(prix)
```

```
{'carottes': 1.49, 'brocolis': 2.49, 'pommes': 0.99, 'avocats': 4.99}
```

[ ]:
```
prix.pop("carottes")
print(prix)
```

```
{'brocolis': 2.49, 'pommes': 0.99, 'avocats': 4.99}
```

2.2.2 3.4.2 Parcours d'un dictionnaire

[ ]:
```
for clef in prix.keys():
    print(clef)
```

```
brocolis
pommes
avocats
```

[ ]:
```
for clef in prix:
    print(clef)
```

```
brocolis
pommes
avocats
```

[ ]:
```
for valeur in prix.values():
        print(valeur)
```

```
2.49
0.99
4.99
```

[ ]:
```
for clef, valeur in prix.items():
        print("Le prix des {} est de {}€/kg".format(clef,valeur))
```

```
Le prix des brocolis est de 2.49€/kg
Le prix des pommes est de 0.99€/kg
Le prix des avocats est de 4.99€/kg
```

[ ]:
```

```

3 4. Programmation dynamique

3.1 4.1. Triangle de Pascal

3.1.1 Calcul récursif naïf des coefficients binomiaux

[ ]:
```
def mesure_duree(f,arg):
    """
    retourne le résultat d'exécution de la fonction f avec arg passé en argument
    """
    t1=time.perf_counter()
    res=f(*arg)
    t2=time.perf_counter()
    print("C{}={} effectué en {:.1e}s".format(arg,res,t2-t1))
```

[ ]:
```
def binom(n, p):
    if p == 0 or n == p:
        return 1
```

```
    return binom(n-1, p-1) + binom(n-1, p)

mesure_duree(binom,(8,2))
```

```
C(8, 2)=28 effectué en 2.1e-05s
```

[ ]:
```
mesure_duree(binom,(8,4))
```

```
C(8, 4)=70 effectué en 2.0e-05s
```

[ ]:
```
mesure_duree(binom,(30,15))
```

```
C(30, 15)=155117520 effectué en 2.0e+01s
```

3.1.2 Version itérative du calcul des coefficients binomiaux

[ ]:
```
def binom_it(n, p):
    t = pl.zeros((n + 1, p + 1))
    for i in range(0, n + 1):
        t[i, 0] = 1
    for i in range(1, p + 1):
        t[i, i] = 1
    for i in range(2, n + 1):
        for j in range(1, min(p, i) + 1):
            t[i, j] = t[i - 1, j - 1] + t[i - 1, j]
    return t[n, p]
```

[ ]:
```
mesure_duree(binom_it,(300,150))
```

```
C(300, 150)=9.375970277282748e+88 effectué en 2.6e-02s
```

3.1.3 Version récursive avec memoisation

[ ]:
```
binom_dict = {}

def binom_mem(n, p):
    if (n, p) not in binom_dict:
        if p == 0 or n == p:
            binom_dict[(n, p)] = 1
        else:
            binom_dict[(n, p)] = binom_mem(n - 1, p - 1) + binom_mem(n - 1, p)
    return binom_dict[(n, p)]
```

[ ]:
```
mesure_duree(binom_mem,(30,15))
```

```
C(30, 15)=155117520 effectué en 2.8e-04s
```

Remarque culturelle Le code suivant permet d'associer automatiquement un dictionnaire à la fonction récursive utilisée.

[ ]:
```
from functools import lru_cache

@lru_cache
def binom(n, p):
    if p == 0 or n == p:
        return 1
    return binom(n-1, p-1) + binom(n-1, p)

mesure_duree(binom,(30,15))
```

```
C(30, 15)=155117520 effectué en 2.5e-04s
```

3.2 4.2. Algorithme de Floyd-Warshall

[ ]:
```
def floydwarshall(M):
    n = M.shape[0]
    N = M.copy()
    for k in range(n):
        for i in range(n):
            for j in range(n):
                N[i, j] = min(N[i, j], N[i, k] + N[k, j])
    return N
```

[ ]:
```
M=pl.array([
[0,float('inf') , float('inf') , float('inf') , -1 , float('inf')],
[1,0,float('inf'),2,float('inf'),float('inf')],
[float('inf'),2,0,float('inf'),float('inf'),6],
[-3,float('inf'),float('inf'),0,float('inf'),float('inf')],
[float('inf'),7,float('inf'),4,0,float('inf')],
[float('inf'),5,-4,float('inf'),float('inf'),0]
])
M
```

[ ]:
```
array([[ 0., inf, inf, inf, -1., inf],
       [ 1.,  0., inf,  2., inf, inf],
       [inf,  2.,  0., inf, inf,  6.],
       [-3., inf, inf,  0., inf, inf],
       [inf,  7., inf,  4.,  0., inf],
       [inf,  5., -4., inf, inf,  0.]])
```

[ ]:
```
floydwarshall(M)
```

[ ]:
```
array([[ 0.,  6., inf,  3., -1., inf],
       [-1.,  0., inf,  2., -2., inf],
       [ 1.,  2.,  0.,  4.,  0.,  6.],
       [-3.,  3., inf,  0., -4., inf],
       [ 1.,  7., inf,  4.,  0., inf],
       [-3., -2., -4.,  0., -4.,  0.]])
```

Complément : fermeture transitive d'un graphe Considérons de nouveau un graphe non pondéré, orienté ou non. Le problème de la fermeture transitive consiste à déterminer si deux sommets a et b peuvent être reliés par un chemin allant de a à b. Pour le résoudre, nous allons utiliser la matrice d'adjacence associée à ce graphe, mais cette fois en utilisant les valeurs booléennes `True` pour dénoter l'existence d'une arête et `False` pour en marquer l'absence. Remplaçons maintenant dans l'algorithme de Floyd-Warshall la relation de récurrence sur les coefficients des matrices $M^{(k)}$par:

$$m_{ij}^{(k+1)} = m_{ij}^{(k)} \text{ ou } \left(m_{i,k+1}^{(k)} \text{ et } m_{k+1,j}^{(k)}\right)$$

On peut prouver de façon similaire à ce que nous avons fait que le booléen $m_{ij}^{(k)}$ dénote l'existence d'un chemin reliant les sommets v_i et v_j en ne passant que par les sommets $v_1, v_2, \ldots, v_k$. a matrice $M^{(n)}$ résout le problème de la fermeture transitive.

L'algorithme ainsi modifié est connu sous le nom d'algorithme de Warshall :

[ ]:
```
def warshall(M):
    n = M.shape[0]
    N = M.copy()
    for k in range(n):
        for i in range(n):
            for j in range(n):
                N[i, j] = N[i, j] or N[i, k] and N[k, j]
    return N
```

[ ]:
```
N=M.copy()
n = M.shape[0]
for i in range(n):
    for j in range(n):
        if N[i,j]==float('inf'):
            N[i,j]=False
        else:
            N[i,j]=True
warshall(N)
```