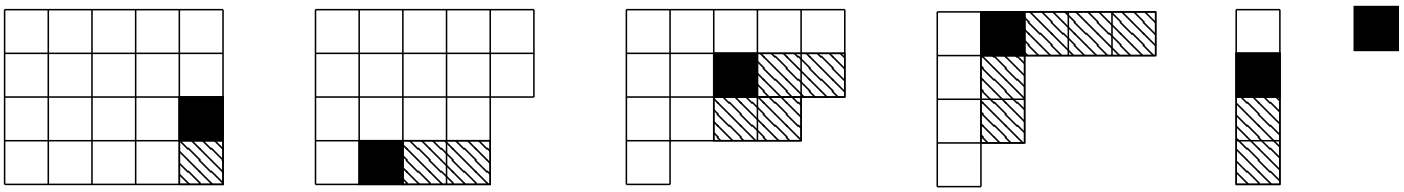
I Jeu de Chomp

Le jeu de Chomp se joue à deux joueurs (Alice et Bob, où Alice commence) sur une tablette de chocolat. Tour à tour, chaque joueur choisit un carré restant de la tablette et le mange ainsi que tous les carrés à droite et en bas de celui-ci. Le joueur qui mange le carré en haut à gauche a perdu. Voici un exemple de partie, où le carré choisi à chaque tour est en noir et les (autres) cases mangées hachurées :



On représente la tablette de chocolat par une matrice, où un 1 indique que la case est mangée et un 0 qu'elle n'est pas mangée. La case en haut à gauche de la tablette de chocolat a pour coordonnées $(0, 0)$ dans cette matrice.

1. Écrire une fonction `grille(n, p)` renvoyant une matrice à n lignes et p colonnes remplie de 0.

Solution :

```
def grille(n, p):
    return [[0]*p for i in range(n)]

def grille(n, p): # autre solution
    G = []
    for i in range(n):
        G.append([0]*p)
    return G

def grille(n, p): # autre solution
    G = []
    for i in range(n):
        G.append([])
        for j in range(p):
            G[i].append(0)
    return G
```

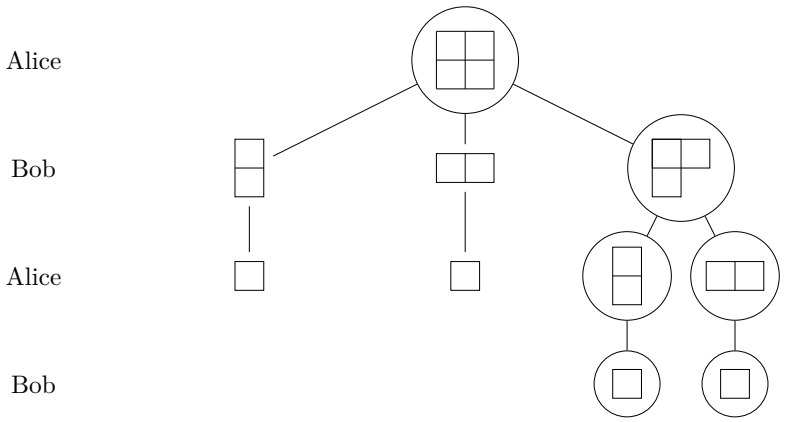
2. Écrire une fonction `coup(g, i, j)` qui modifie la matrice g en jouant un coup sur la ligne i et la colonne j.

Solution :

```
def coup(g, i, j):
    for k in range(i, len(g)):
        for l in range(j, len(g[0])):
            g[k][l] = 1
```

3. Dessiner le graphe des configurations pour une tablette initiale de taille 2×2, c'est-à-dire le graphe dont les sommets sont les configurations possibles de la tablette et dont les arêtes sont les coups possibles. On indiquera sur chaque sommet si c'est Alice ou Bob qui doit jouer.

Solution : On indique à gauche le joueur qui doit jouer.



4. Déterminer, à la main, les attracteurs pour Alice sur une tablette initiale de taille 2×2. On pourra les indiquer sur le graphe précédent.

   Solution : Les attracteurs sont entourés sur le graphe précédent. On les détermine de bas en haut, comme dans le cours.

Mines Ponts 2021 (les 3 marches) : éléments de correction

Entrée [1]:
```
%matplotlib inline
```

Partie I. Randonnée

Q1 Nombre de participants nés entre 1999 et 2003 (inclus) :

```
SELECT COUNT(*) FROM Participant
WHERE ne >= 1999 AND ne <= 2003
```

Q2 Durée moyenne des randonnées pour chaque niveau de difficulté :

```
SELECT diff,AVG(duree) FROM Rando
GROUP BY diff
```

Q3 Nom des participants pour lesquels la randonnée n°42 est trop difficile :

- avec une sous-requête :

```
SELECT pnom FROM Participant
WHERE diff_max < (SELECT diff FROM Rando
                  WHERE rid=42)
```

- avec un produit cartésien :

```
SELECT pnom FROM Participant,Rando
WHERE rid = 42 AND diff_max < diff
```

Q4 Clés primaires des randonnées qui ont un ou des homonymes, sans redondance :

- Première version, avec une auto-jointure :

```
SELECT DISTINCT R.rid
FROM Rando AS R JOIN Rando AS S
ON R.rnom=S.rnom
WHERE R.rid <> S.rid
```

- Deuxième version, avec un `GROUP BY` et un `HAVING` :

```
SELECT DISTINCT rid FROM Rando
WHERE rnom IN (SELECT rnom FROM rando
               GROUP BY rnom
               HAVING COUNT(*) > 1)
```

Q5 De la lecture de fichier :

- Une première version :

```
def importe_rando(nom_fichier):
    fichier=open(nom_fichier,"r")
    coords=[]
    fichier.readline() # pour ne pas traiter la 1ère ligne
    for ligne in fichier:
        ligne=ligne.split(",")
        ligne=[float(elt) for elt in ligne]
        coords.append(ligne)
    fichier.close()
    return coords
```

- On peut aussi écrire une fonction plus "condensée" :

```
def importe_rando(nom_fichier):
    fichier=open(nom_fichier,"r")
    fichier.readline() # pour ne pas traiter la 1ère ligne
    coords=[[float(elt) for elt in ligne.split(",")] for ligne in fichier]
    fichier.close()
    return coords
```

- On pourrait également utiliser `readlines`, ou utiliser une syntaxe de style `with open(nom_fichier,"r") as fichier:`

Q6 C'est une recherche de maximum :

```
def plus_haut(coords):
    lat,long,m=coords[0][:3]
    for elt in coords[1:]:
        if elt[2] > m:
            lat,long,m = elt[:3]
    return [lat,long]
```

Une variante :

```
def plus_haut(coords):
    pos=0
    m=coords[pos][2]
    for i in range(1,len(coords)):
        cur=coord[i][2]
        if cur > m:
            m=cur
            pos=i
    return coords[pos][:2]
```

NB : Si plusieurs points ont la même altitude maximale, la fonction précédente renvoie le premier point de la liste qui atteint cette attitude.

Q7 On propose deux versions :

- Si on ne s'autorise pas la commande `sum` :

```
def deniveles(coords):
    pos,neg=0,0
    for i in range(len(coords)-1):
        a,b=coords[i][2],coords[i+1][2]
        if a>b: # dénivelé négatif
            neg+=b-a
        else: # dénivelé positif
            pos+=b-a
    return [pos,neg]
```

- si on s'autorise la commande `sum` :

```
def deniveles(coords):
    pentes=[coords[i+1][2]-coords[i][2] for i in range(len(coords)-1)]
    pos=sum([p for p in pentes if p>0])
    neg=sum([p for p in pentes if p<0])
    return [pos,neg]
```

Q8 On importe les fonctions utiles du module `math` :

```
from math import asin,sin,cos,sqrt,radians

RT = 6371 # variable globale donnée dans le canevas

def distance(c1,c2):
    phi1,l1,alt1=c1[:3] # phi,lambda et altitude pour le point c1
    phi2,l2,alt2=c2[:3] # idem pour c2
    phi1,phi2=radians(phi1),radians(phi2) # conversion en radians
    l1,l2=radians(l1),radians(l2)
    alt=RT*1e3+(alt1+alt2)/2 # conversion en mètres de RT + on rajoute l'altitude moyenne
    s=sin((phi2-phi1)/2)**2+cos(phi1)*cos(phi2)*sin((l2-l1)/2)**2
    s=sqrt(s)
    d=2*alt*asin(s) # formule de haversine
    dis=sqrt(d**2+(alt2-alt1)**2) # théorème de Pythagore
    return dis
```

Q9 Calcul classique de somme :

- sans utiliser `sum` :

```
def distance_totale(coords):
    d=0
    for i in range(len(coords)-1):
        d+=distance(coords[i],coords[i+1])
    return d
```

- en utilisant `sum` :

```
def distance_totale(coords):
    dis=sum(distance(coords[i],coords[i+1]) for i in range(len(coords)-1))
    return dis
```

Partie II. Mouvement brownien d'une petite particule

Q10

```
def vma(v1,a,v2):
    assert len(v1) == len(v2) # vérification de la longueur identique des listes
    return [v1[i] + a * v2[i] for i in range(len(v1))]
```

Q11 On projette l'équation du mouvement sur l'axe des abscisses ; on obtient :

$$\ddot{x} = -\frac{\alpha \dot{x}}{m} + \frac{f_{Bx}}{m}$$

On a une relation identique en projetant sur l'axe des ordonnées.

Ne pas oublier l'import du module `random` :

```
from math import cos,sin,pi # ou alors déjà fait à la question 8
import random as rd

def derive(E):
    x,y,xp,yp = E
    theta = rd.uniform(0,2*pi)
    norme = abs(rd.gauss(MU,SIGMA))
    fBx = cos(theta)*norme # on projette fB sur (Ox)
    fBy = sin(theta)*norme # idem sur (Oy)
    xpp = (-ALPHA*xp + fBx)/M
    ypp = (-ALPHA*yp + fBy)/M
    return [xp,yp,xpp,ypp]
```

Q12 La relation de récurrence produite par la méthode d'Euler (explicite) est la suivante :

$$\frac{E_{n+1} - E_n}{dt} = \dot{E_n}$$

i.e. :

$$E_{n+1} = E_n + dt \times \dot{E_n}$$

On en déduit la fonction suivante :

```
def euler(E0,dt,n):
    Es = [E0]
    E = E0
    for i in range(n):
        E = vma(E,dt,derive(E)) # relation de récurrence d'Euler
        Es.append(E)
    return Es
```

Partie III. Marche auto-évitante

Q13 On parcourt les voisins du point (x, y) pour voir ce qui ont déjà été atteints par le chemin :

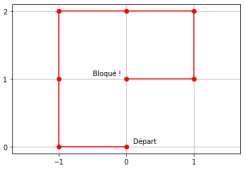
```
def positions_possibles(p,atteints):
    possibles = []
    x,y = p
    voisins = [[x+1,y],[x-1,y],[x,y+1],[x,y-1]] # les 4 voisins de (x,y)
    for v in voisins:
        if not(v in atteints):
            possibles.append(v)
    return possibles
```

Q14 Il suffit de tourner en "escargot" pour s'enfermer dans une impossiblité le plus rapidement possible.

Remarque : le code n'était évidemment pas demandé dans l'énoncé !

Entrée [2]:

```
import matplotlib.pyplot as plt
chemin_x = [0,-1,-1,-1,0,1,1,0]
chemin_y = [0,0,1,2,2,2,1,1]
plt.plot(chemin_x,chemin_y,"o-r")
plt.axis("equal")
plt.grid("on")
plt.xticks([-1, 0, 1])
plt.yticks([0, 1, 2])
plt.text(0.1, 0.05, "Départ")
plt.text(-0.5, 1.05, "Bloqué !")
plt.show()
```



Il s'agit du plus court chemin auto-bloquant. Il est de longueur 7. Il y en a 8 en tout, que l'on peut déduire par 4 rotations (d'angle droit et de centre $(0, 0)$) composées ou non par une symétrie par rapport à (Ox).

Q15 On n'oublie pas les imports (s'ils n'ont pas déjà été faits) :

```
import random as rd

def genere_chemin_naif(n):
    chemin = [[0,0]]
    p=[0,0]
    for i in range(n): # il faut n+1 points pour un chemin de longueur n
        possibles = positions_possibles(p,chemin)
        if len(possibles) == 0:
            return None # chemin auto-bloquant
        else:
            p = rd.choice(possibles)
            chemin.append(p)
    return chemin
```

Q16 Dans le pire des cas (aucun blocage), il y aura n appels à la fonction `positions_possibles` , avec comme paramètre `chemin` , une liste de taille s'incrémentant de 1 à chaque étape (puisqu'on ne bloque jamais). Chaque appel coûte 4 tests d'appartenance d'une liste de longueur 2 à la liste de telles listes `chemin` , soit un $O($ `len(chemin)` $)$ comparaisons.

Cela donne donc une complexité d'ordre $O(1 + 2 + \cdots + n) = \left(\frac{n(n+1)}{2}\right) = O(n^2)$ comparaisons.

N.B. : on a négligé les autres opérations (coût du `choice` , du `append` , affectations).

Q17

- La boucle intérieure `for i in range(1,M):` détermine la fréquence d'apparition de chemins auto-bloquants de taille fixée `n` parmi $N = 10000$ chemins de longueur n générés aléatoirement.
- La boucle extérieure `for n in range(1,M):` effectue ce calcul de fréquence pour toutes les longueurs de chemins entre 1 et $M - 1 = 350$.
- Il s'agit donc, pour chaque longueur $n \in [\![1, 350]\!]$, de donner une approximation de la probabilité de générer un chemin auto-bloquant de taille n.
- Conclusion : on a donc tracé une approximation de la probabilité pour un chemin de longueur n d'être auto-bloquant en fonction de n. Cette probabilité semble croître (ce qui est normal pour un algorithme glouton) vers 1 : plus n est grand, moins il est probable de générer un CAE de longueur n par la méthode naïve (et le calcul de complexité quadratique précédent ne sera pas pertinent).

Q18 Le tri-fusion réalise un tri d'une liste de taille n, dans le cas le pire, avec une complexité en $O(n \ln n)$. C'est la meilleure complexité possible dans le cas le pire.

Attention : Dans le cas le pire, le tri rapide a une complexité quadratique.

Q19 Une fois la liste des points triée, il suffit de regarder si deux points consécutifs sont égaux :

```
def est_CAE(chemin):
    chemin_trie=sorted(chemin)
    for i in range(len(chemin)-1):
        if chemin_trie[i]==chemin_trie[i+1]:
            return False
    return True
```

Si le chemin est de taille n, le tri coûte $O(n \ln n)$ opérations, et la boucle coûte n accès et tests d'égalité entre deux points, qui se font en $O(1)$ opérations (complexité amortie ?).

La complexité dans le pire des cas de la fonction précédente est bien en $O(n \ln n)$.

Q20 On suppose que l'on a orienté le plan dans le sens trigonométrique.

```
def rot(p,q,a):
    x,y = p
    u,v = q
    assert(a in [0,1,2])
    if a==0:
        return [2*x-u,2*y-v]
    elif a==1:
        return [x+y-v,y+u-x]
    else:
        return [x+v-y,y+x-u]
```

Q21 On propose 2 versions :

- Avec des boucles :

```
def rotation(chemin,i_piv,a):
    chemin_piv=[]
    for i in range(i_piv+1):
        chemin_piv.append(chemin[i])
    p=chemin[i_piv]
    for i in range(i_piv+1,len(chemin)):
        chemin_piv.append(rot(p,chemin[i],a))
    return chemin_piv
```

- Avec les listes par compréhension :

```
def rotation(chemin,i_piv,a):
  debut=chemin[:i_piv+1] # le pivot est invariant par rotation
  p=chemin[i_piv]
  fin=[rot(p,q,a) for q in chemin[i_piv+1:]]
  return debut+fin
```

Q22 À chaque étape, on génère un nouveau chemin pivoté jusqu'à ce que l'on trouve un CAE :

```
import random as rd # si on ne l'a pas déjà fait avant

def genere_chemin_pivot(n,n_rot):
    chemin = [[i,0] for i in range(n+1)] # intialisation donnée dans l'énoncé
    for i in range(n_rot):
        bloque = True
        while bloque: # tant que l'on génère un chemin qui bloque
            i_piv = rd.randrange(1,n)
            a = rd.randrange(0,3)
            chemin_piv = rotation(chemin,i_piv,a) # nouveau chemin potentiel
            if est_CAE(chemin_piv): # le nouveau chemin est un CAE
                chemin = chemin_piv # mise à jour du chemin
                bloque = False # on sort de la boucle while
    return chemin
```

Q23 Si p_{piv}, p, q désignent le pivot, son point précédent et son point suivant, une des 3 rotations (d'angle π, $\frac{\pi}{2}$ ou $-\frac{\pi}{2}$) va envoyer q sur p. Il faut l'éviter (cela gagne un appel -coûteux- à `est_CAE`), et pour cela modifier la ligne `a = rd.randrange(0,3)` de la fonction précédente, et la remplacer par le code suivant (par exemple) :

```
i_piv = rd.ranrange(1,n) # déjà écrit
p_piv=chemin[i_piv]
p=chemin[i_piv-1]
q=chemin[i_piv+1]
det=(q[0]-p_piv[0])*(p[1]-p_piv[1])-(p[0]-p_piv[0])*(q[1]-p_piv[1])
if det==0:
    a=rd.choice([1,2])
elif det>0:
    a=rd.choice([0,2])
else:
    a=rd.choice([0,1])
chemin_piv = rotation(chemin,i_piv,a) # déjà écrit
```

On pourrait également tester les 3 rotations possibles pour éliminer celle qui envoie p sur q.