1. Jeu de Nim

1.
```
def Nim(n,k) :
    return {i : list(range(max(i-k,1),i)) for i in range(1,n+1)}
```

2.
```
def biparti(G) :
    B = {}
    for s in G :
        B[(s,0)] = [(0,t) for t in G[s]]
        B[(0,s)] = [(t,0) for t in G[s]]
    return B
```

3. J_1 gagne si la position initiale `(n,0)` est dans son attracteur, sinon c'est J_2 qui gagne (car il n'y a pas de match nul au jeu de Nim). Le premier coup à jouer pour J_1 est alors un coup accessible et qui reste dans sont attracteur.

a) J_1 gagne et il doit aller en `(0,4)` donc prendre 1 jeton

b) J_2 gagne

c) J_1 gagne et il doit aller en `(0,17)` donc prendre 3 jetons

4. Pour rejoindre la position gagnante depuis `(n,0)`, on a besoin de jouer moins de n coups (en fait au plus $n/2$ pour le joueur 1)

```
def jeu(G,n) :
    strategie = {}
    F = [(0,1)]                 # à atteindre
    A = attracteur(G,F)
    pos1 = [(n,0)]              # positions de J1 après k coups joués
    for _ in range(n) :         # moins de k coup à jouer
        pos2 = []               # les positions que J2 occupera après les dé
            placements faits pas J1
        for p1 in pos1 :        # pour chacunes des positions de J1, on choisit
            un déplacement qui est dans l'attracteur
            depl1 = G[p1]
            for k in depl1 :
                if k in A :
                    p2 = k
            strategie[p1] = p2  # on construit la stratégie avec un des ces
                coups
            pos2.append(p2)
        pos1 = []
        for p2 in pos2 :        # pour chaque position où on a placé J2, il
            faut prévoir tous les coups qu'il peut faire
            depl2 = G[p2]
            pos1 += depl2       # et on recommence avec le résultat de tous
                les coups qu'il peut jouer
    return strategie
```

Pour une fonction plus efficace en terme de complexité, il serait préférable de construire la stratégie en même temps que l'on détermine l'attracteur.

2. Positions finale

```
def Final(G,k) :
    F = []
    for s in G :
        if s[k-1] == 0 and len(G[s]) == 0 :
            F.append(s)
    return F
```

3. Attracteur récursif

```
def att(G,F) :
    TG = transp(G)
    test = {s : False for s in G}
    # la fonction récursive calcule l'ensemble Ai en modifiant le dictionnaire
        test (donc pas de return nécessaire)
    def A(G,TG,i,F) :
        if i == 0 :                          # on initialise test avec A0=F
            for s in F :
                test[s] = True
        else :
            A(G,TG,i-1,F)                    # on calcule A[i-1] (en modifiant test)
            for (i,j) in G :
                if i == 0 and test[(i,j)] : # (i,j) contrôlée par J2 dans A[i-1]
                    for t in TG[(i,j)] :     # elles conviennent toutes
                        test[t] = True
                elif test[(i,j)] :           # (i,j) contrôlée par J1 dans A[i-1]
                    for t in TG[(i,j)] :     # on cherche celles qui conviennent
                        ok = True
                        for u in G[t] :
                            ok = ok and test[u]
                        if ok :
                            test[t] = True
    A(G,TG,len(G),F)
    return [s for s in test if test[s]]
```