TP 1 python - IHM et tkinter

les tours de Hanoï

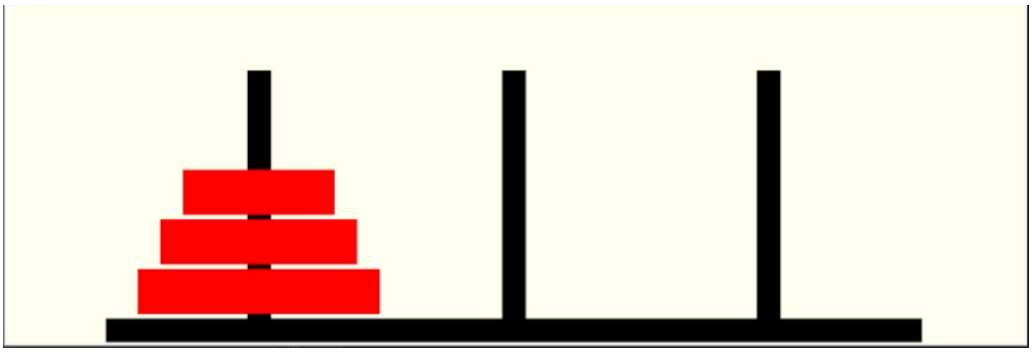
Nom : KESKES Nazim
Groupe: A

2.2) L'algorithme de résolution récursif :

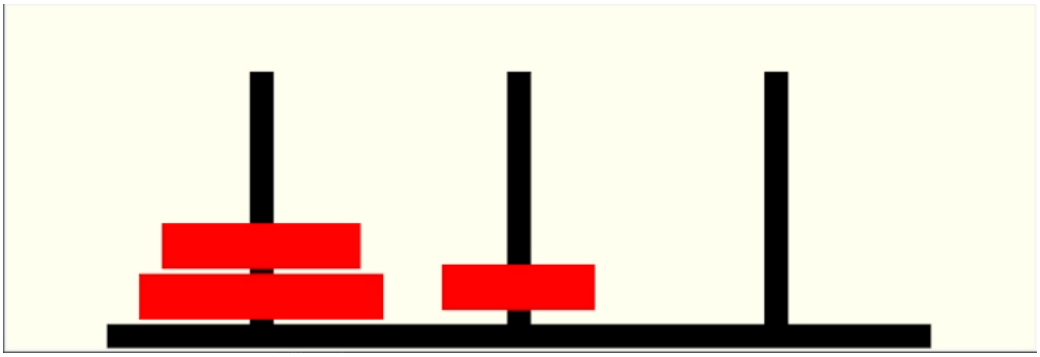
2.2.1) la résolution du problème pour 3 disques :

Afin de bien comprendre le principe de l'algorithme voici les différents déplacements des 3 disques :

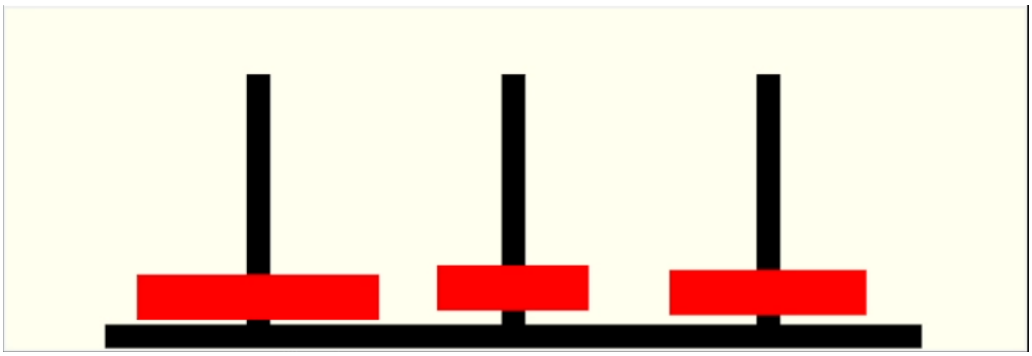
Cas initial :



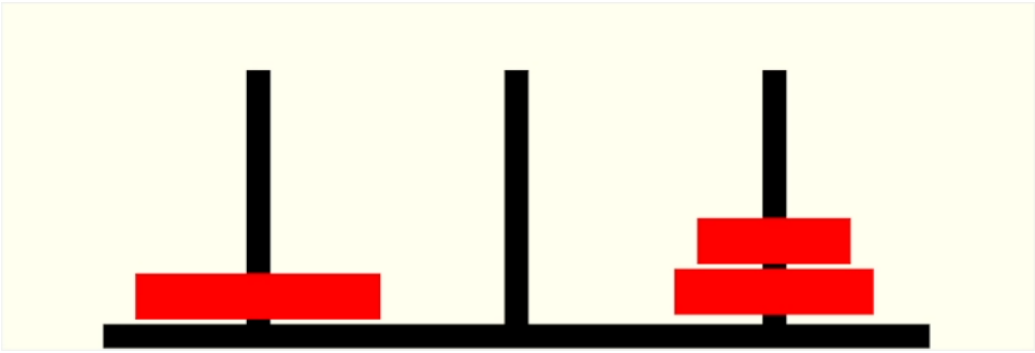
Deplacement 1 :



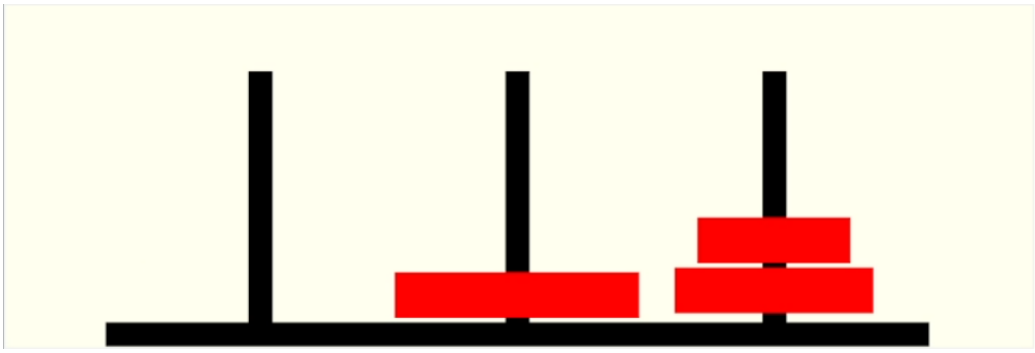
Deplacement 2 :



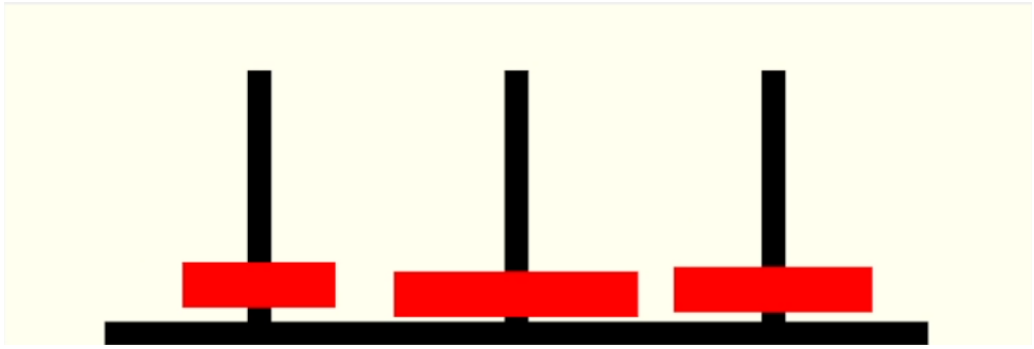
Deplacement 3 :



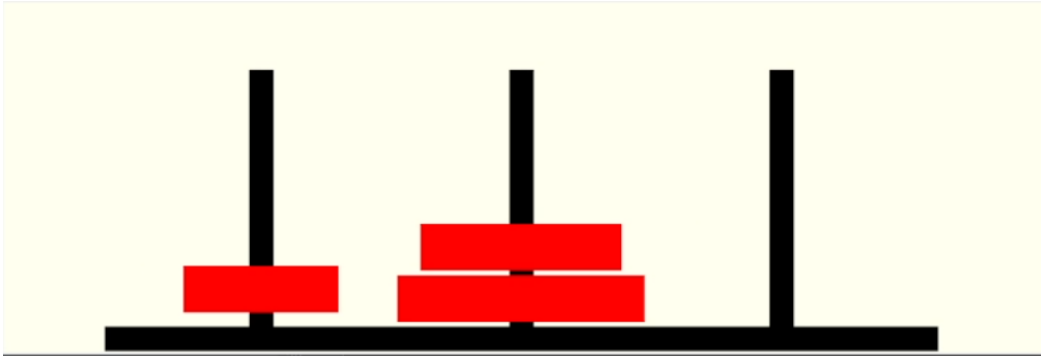
Deplacement 4 :



Deplacement 5 :



Deplacement 6 :



Deplacement 7 :

On remarque qu' il y a 7 déplacements.
On note les différentes tiges par A, B et C. Pour bien comprendre le fonctionnement, on note chaque déplacement par le tuple **(source, destination).** Donc notre suite se résume par : (A,B) , (A,C) , (B,C) , (A,B) , (C,A) , (C,B) , (A,B) .

On considère que la tige intermédiaire est la tige C.
Si on pose **n** disques. On remarque que le déroulement se déroule dans 3 phases principales :

- Déplacer tous les disques de la tige source sauf le dernier vers la tige intermédiaire.
- Placer ce dernier disque vers sa destination.
- Déplacer les n - 1 disques de la tige intermédiaire vers la tige destination.

Dans la 1ere et 3eme étape, pour déplacer les **n - 1** disques (c-à-d sauf le dernier) , on remarque qu'on résout le même problème à savoir déplacer un certain nombre de disques d'une tige à une autre en en disposant du 3e comme tige intermédiaire.
Par la suite, c'est un problème récurrent ou a chaque fois, on applique les 3 étapes précédentes et on passe a un problème plus restreint et enfin on change les 3 tiges (source, destination, intermédiaire).

2.2.2) la fonction hanoi(n,src,des,tmp) :

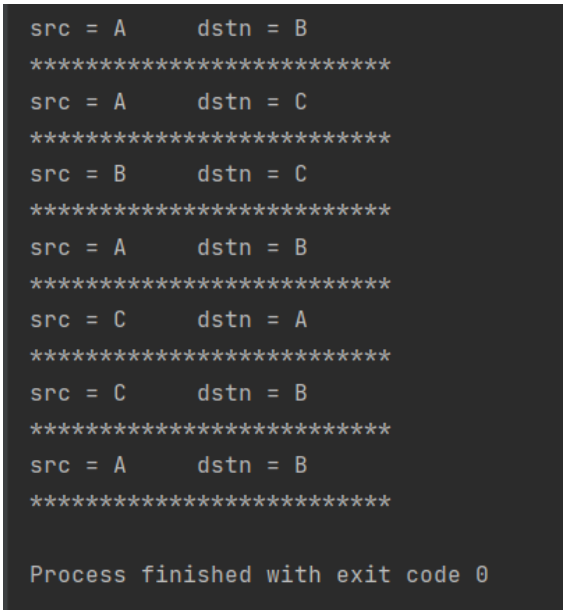
La fonction est en pièce jointe dans le dossier **src** sous le nom de hanoi.py.

```python
def hanoi(n, src, des, tmp):
    if n > 0:
        hanoi(n - 1, src, tmp, des)
        print("src =", src, "\tdst =", des, "\n**************************")
        hanoi(n - 1, tmp, des, src)

hanoi(3, "A", "B", "C")
```

2.2.3) le déroulement de l'exécution de la fonction :

Si on execute **hanoi(3, "A", "B", "C")** on obtient :



On remarque qu' on est tombé dans la même suite que celle de la question 1.
Pour généraliser le cas (voir le code dans la pièce jointe) :

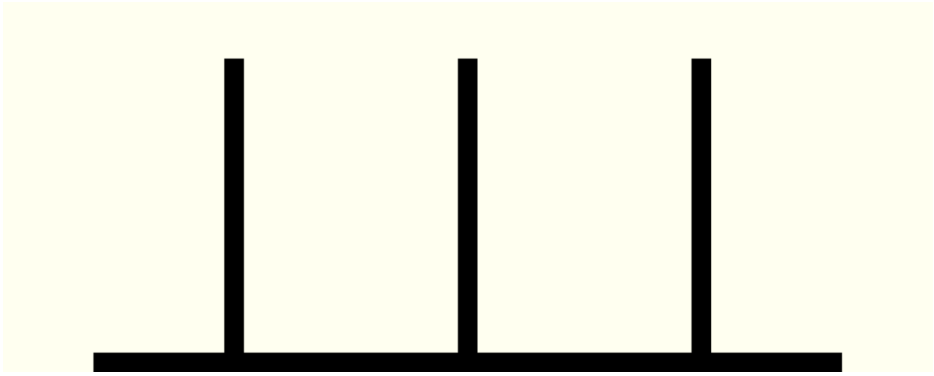
- on réalise hanoi(n, X, Y, Z)
- lorsqu'on déplace les n−1 tiges de X vers la tige Y, on fait l'appel hanoi(n - 1, X, Z, Y)
- lorsqu'on déplace les n−1 tiges de Z vers la tige Y, on fait l'appel hanoi(n - 1, Z, Y, X)

Dans le cas arrivant a n = 0 , on commence à remonter dans la pile de récursivité dans cette ordre :
hanoi(1,"A","B", "C")
hanoi(2,"A","C", "B")
hanoi(1,"B","C", "A")
hanoi(3,"A","B", "C")
hanoi(1,"C","A", "B")
hanoi(2,"C","B", "A")
hanoi(1,"A","B", "C")
à chaque fois qu'on exécute la procédure **afficher(print)** , on aura donc : (A,B) , (A,C) , (B,C) , (A,B) , (C,A) , (C,B) , (A,B) .

2.3) Dessiner le décor en Tkinter:

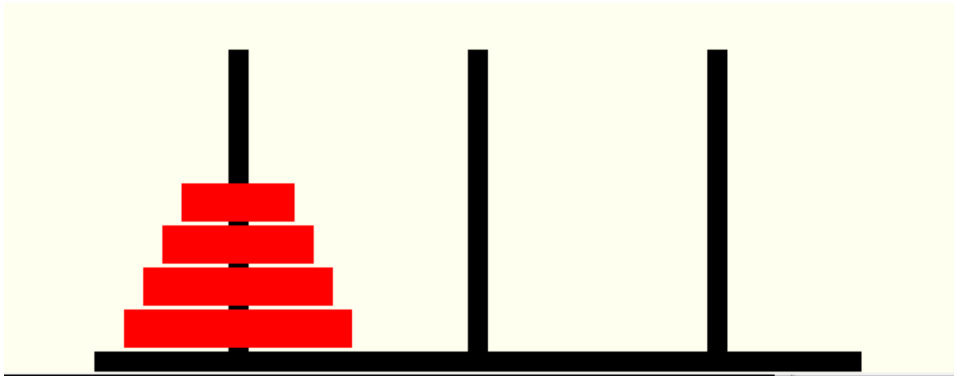
2.3.1) Dessiner le canevas, les trois tiges et le socle :

Le code est en pièce jointe dans le dossier **src** sous le nom de caneva.py. Voici le résultat de l'exécution :



2.3.2) placer les disques sur la tige de gauche :

Le code est en pièce jointe dans le dossier **src** sous le nom de caneva_avec_disque.py. Voici le résultat de l'exécution :

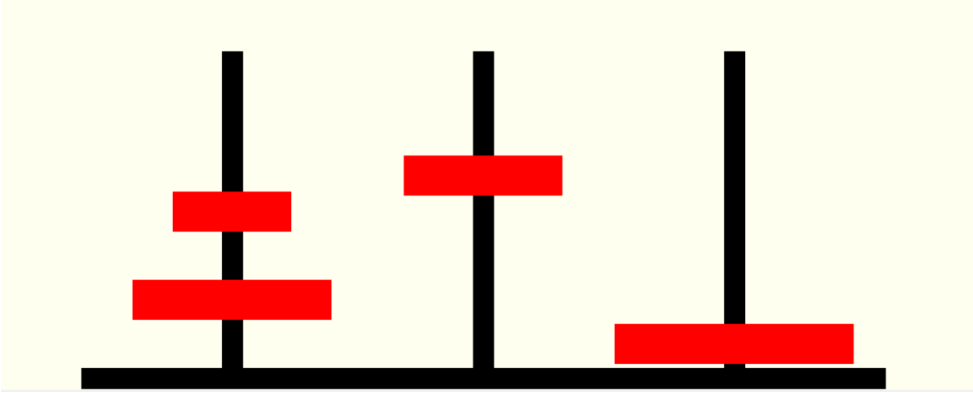


2.4) Déplacement d'un disque move(nro,src,des):

```python
#Déplacement d'un disque
def move(nro, src, des):
    i, hi = src
    j, hj = des
    dx = (j - i) * Distance
    dy = -(hj - hi) * Height
    cnv.move(ids[nro], dx, dy)

#test1 deplacement
move(2, (0,2), (1, 4))
#test2 deplacement
move(0, (0,0), (2, 0))
```

Le code est en pièce jointe dans le dossier **src** sous le nom de deplacement.py. Voici le résultat de l'exécution d'un test en déplaçant le disque[0] a la tige C avec la même position initial et aussi le disque[2] a la tige B avec la 4eme position :



2.2.3) Adaptation de l'algorithme pour Tkinter :

2.5.1) La fonction hanoi(ids,source,destination,temp,hauteurs):

```
def hanoi(ids, source, destination, temp, hauteurs):
    if ids:
        hanoi(ids[1:], source, temp, destination, hauteurs)
        print(ids[0], (source, hauteurs[source] - 1),
              (destination, hauteurs[destination]))
        hauteurs[source] -= 1
        hauteurs[destination] += 1
        hanoi(ids[1:], temp, destination, source, hauteurs)

#test1 de l'algo
hanoi([0, 1, 2], 0, 2, 1, [3, 0, 0])
```

Le code est en pièce jointe dans le dossier **src** sous le nom de hanoi_adapte.py.

2.5.2) le déroulement de l'exécution de la fonction :

Si on execute **hanoi([0,1,2], 0, 2, 1, [3,0,0])** on obtient :

```
2 (0, 2) (1, 0)
1 (0, 1) (2, 0)
2 (1, 0) (2, 1)
0 (0, 0) (1, 0)
2 (2, 1) (0, 0)
1 (2, 0) (1, 1)
2 (0, 0) (1, 2)
```

Maintenant à la place de n (nombre de disques) on travaille directement avec la liste des disques vu que les rectangles sont déjà construits.Ce qui en plus par rapport à la première version est la hauteur dans la tige pour savoir ou on doit placer le disque après avoir le déplacer

(sur quel niveau sur la tige correspondante). Pour généraliser le cas (voir le code dans la pièce jointe) :

- on réalise hanoi(ids, source, destination, temp ,hauteurs)
- lorsqu'on déplace les n−1 tiges de source vers la tige destination, on fait l'appel hanoi(ids[1:], source, tmp, destination , hauteurs)
- lorsqu'on déplace les n−1 tiges temp vers la tige destination, on fait l'appel hanoi(ids[1:], temp, destination, source, hauteurs)

Dé que on effectue un déplacement , on diminue la hauteur de la tige source et on l'augmente dans la tige destination

Dans le cas arrivant ou la liste des disque (ids) est vide , on commence à remonter dans la pile de récursivité dans cette ordre :
hanoi([2],0,1,2,[3,0,0])
hanoi([1,2],0,2,1,[2,1,0])
hanoi([2],1,2,0,[1,1,1])
hanoi([0,1,2],0,1,2,[1,0,2])
hanoi([2],2,0,1,[0,1,2])
hanoi([1,2],2,1,0,[1,1,1])
hanoi([2],0,1,2,[1,2,0])

Par la suite, à chaque fois qu'on exécute la procédure **afficher(print)** , on aura donc : le disque cible , son déplacement d'une tige avec une hauteur vers une autre tige avec son hauteur.

2.6) Animation pour n disques :

2.6.1) la fonction hanoi(ids,source,destination,temp,hauteurs,done):

Le code est en pièce jointe dans le dossier **src** sous le nom de hanoi_version_final.py.

```python
#Realiser l'algorithme de hanoi
def hanoi(ids, source, destination, temp, hauteurs, done):
    if ids:
        A = hanoi(ids[1:], source, temp, destination, hauteurs, False)
        B = [(ids[0], (source, hauteurs[source] - 1), (destination,hauteurs[destination]))]
        hauteurs[source] -= 1
        hauteurs[destination] += 1
        C = hanoi(ids[1:], temp, destination, source, hauteurs, False)
        if done:
            moves = list(enumerate(A + B + C))
            for (i, (nro, src, des)) in moves:
                cnv.after(1000 * (i + 1), move, nro, src, des)
        return A + B + C
    return []

#test hanoi pour n = 4
hanoi(list(range(n)), 0, 1, 2, [n, 0, 0], True)
```

2.6.2) le déroulement de l’exécution de votre fonction:

On remarque que comme un processus, on fait exactement la même chose que la version améliorée. Sauf que cette fois si le fonction **hadoi** renvoie la liste des déplacements. Donc, au lieu d’afficher chaque déplacement d’une manière ascendante dans la récursivité comme la version précédente. Maintenant, on stocke ses déplacements dans une liste moves. Et pour chaque move i (ou déplacement d’un disque) on attribue un temps de lancement qu’il est : (i + 1) secondes. De cette manière, on garantie que les déplacements vont être visualisés graphiquement d’une manière fluide (cad chaque déplacement déroule dans une seconde). Si, on a laissé le code de la version améliorée tel qu’il est, les déplacements se feront instantanément. Donc, chaque déplacement i vas se chevaucher avec le déplacement i - 1. Donc, on ne respecte pas les normes de l'ergonomie de notre animation.