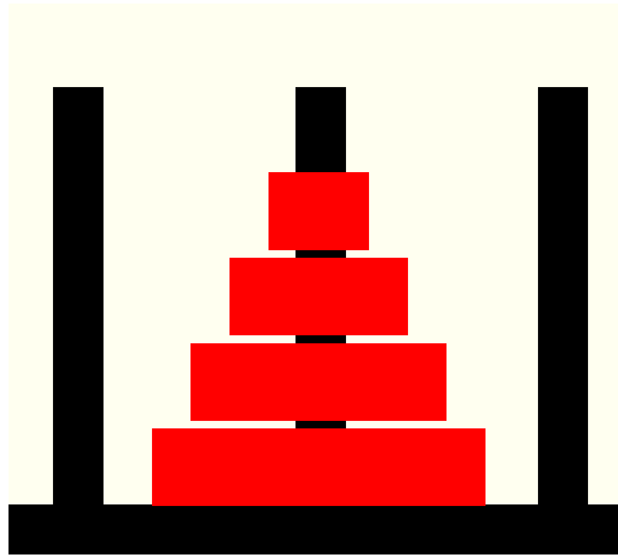
Pascal ORTIZ



Les tours de Hanoï

Version du 8 juin 2021

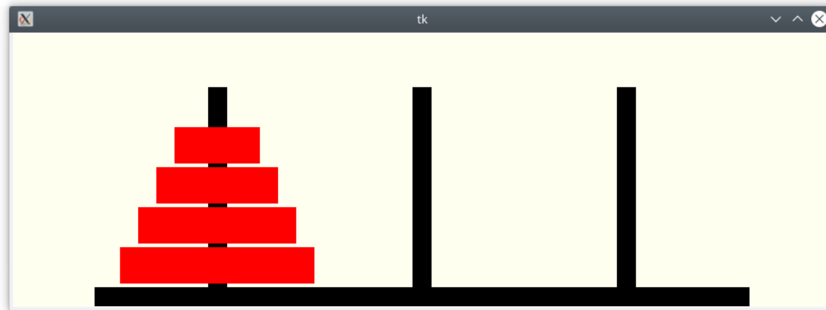
Licence CC-BY

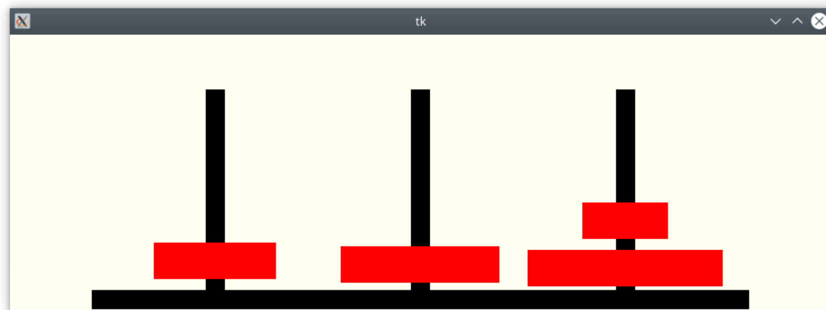
Table des matières

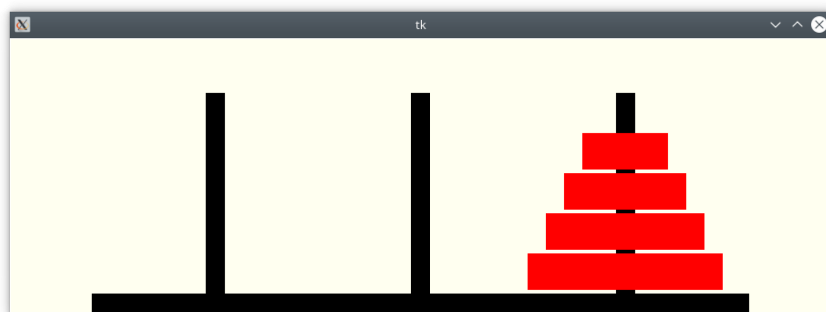
Présentation de l'animation 2
L'algorithme de résolution récursif 3
Dessiner le décor en Tkinter 6
Adaptation de l'algorithme pour Tkinter 10
Déplacement d'un disque 12
Animation pour n disques 15
Prolongements 17

Présentation de l’animation

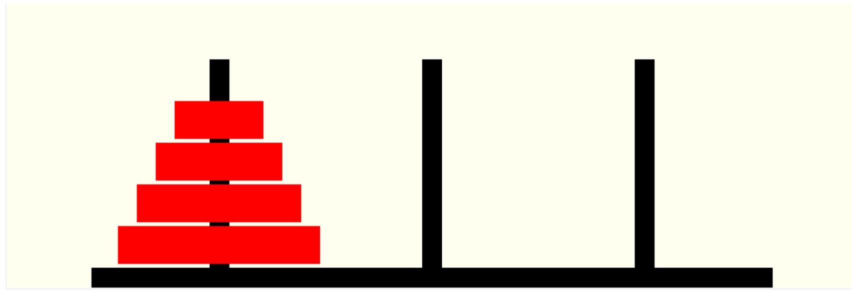
L’activité va consister à écrire en Tkinter l’animation ci-dessous et qui résout l’énigme dite des *tours de Hanoï* :



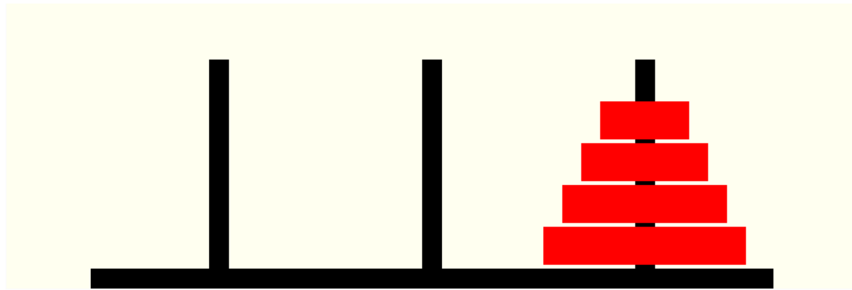




Précisons ce qu’on appelle le problème des tours de Hanoï. On dispose de 3 tiges fixées sur un socle et sur celle de gauche on a disposé, les uns sur les autres, des disques de diamètres croissants et percés :



Le problème consiste à déplacer les disques, un par un, sur la tige de droite :



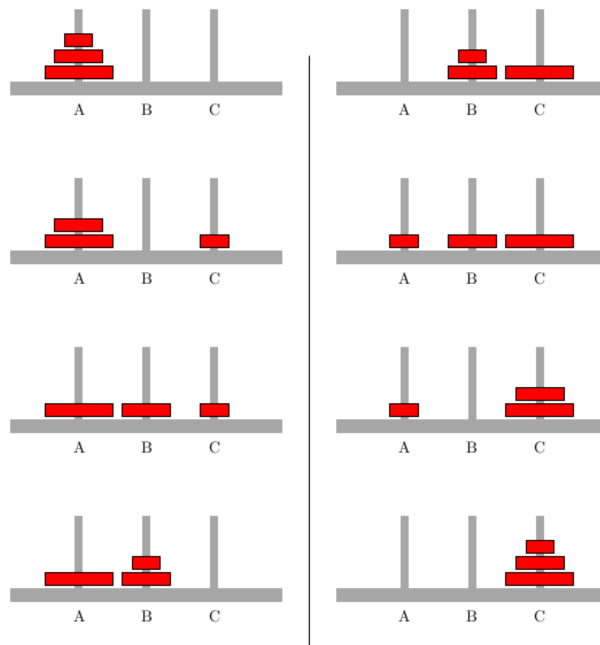
La règle de déplacement d'un disque est la suivante

— le disque est au sommet d'une pile
— on le pose soit sur le socle soit sur un disque de diamètre supérieur

C'est une énigme imaginée au 19e siècle par le mathématicien Édouard Lucas. Un algorithme de résolution du problème ainsi que la réalisation d'une animation sont des questions classiques. Par exemple, l'animation figure dans le code source de CPython en démonstration du module Tkinter et du module Turtle et dans certains ouvrages d'apprentissage de la programmation comme le livre d'Yves Bailly.

L'algorithme de résolution récursif

Dans un premier temps, on ignore complètement l'implémentation en Tkinter, on se concentre sur la solution de l'énigme. Commençons par détailler la résolution du problème pour 3 disques :



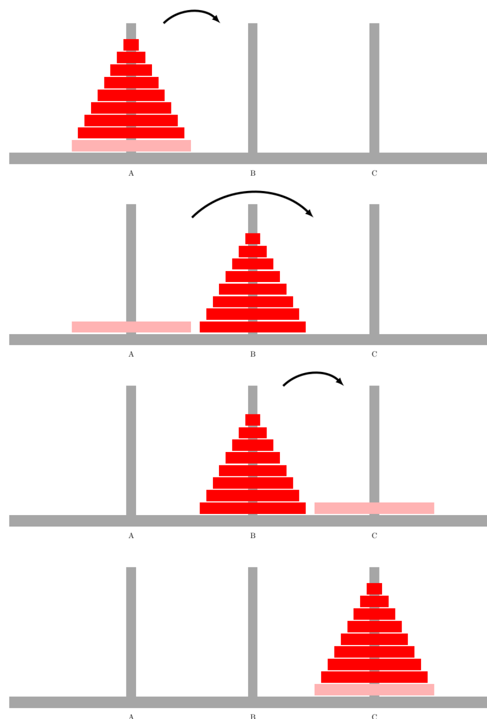
On voit qu'il y a 7 déplacements. Chaque déplacement sera indiqué ci-dessous par un couple (`src`, `dstn`) où `src` est la tige d'origine du disque et `dstn` est la tige de destination. On nommera les tiges A, B et C. Dans le cas de 3 disques, la suite des mouvements est :

(A, C), (A, B), (C, B), (A, C), (B, A), (B, C), (A, C).

Il importe d'avoir en tête, qu'au départ, la tige B joue un rôle de *tige intermédiaire* pour effectuer le déplacement.

Pour la suite, il est **indispensable** d'être familier avec la récursivité, voir par exemple mon tutoriel à ce sujet.

L'idée de base de l'algorithme de déplacement de nos n disques est décrite par les étapes suivantes :



- étape 1 : déplacer tous les disques de la tige A sauf le dernier vers la tige intermédiaire, ce qui libère le dernier disque
- étape 2 : placer ce dernier disque vers sa destination sur la tige C
- étape 3 : de déplacer les disques de la tige B sur la tige C, ce qui est possible vu les diamètres des disques ; pour cela, la tige A servira de tige intermédiaire.

Noter qu'aux étapes 1 et 3, il ne s'agit pas de déplacer **en bloc** la pile de disques (ce n'est pas permis) mais de réaliser la suite des déplacements disque par disque qui conduira au déplacement de la pile de disques indiquée. D'autre part, le point essentiel est que les déplacements des étapes 1 et 3 sont en fait le **même problème** que celui qu'on a résoudre à savoir déplacer un certain nombre de disques d'une tige à une autre en en disposant du 3e comme tige intermédiaire ; le point essentiel ici est qu'il y a **un disque de moins** ($n-1$ au lieu de n) à déplacer.

Précisons la fonction récursive. Il ne faut plus raisonner en termes des tiges initiales A, B et C. Imaginons plutôt que nous soyons dans la situation d'avoir une pile d'au moins n disques sur la tige T (pas forcément la tige A), qu'on veuille déplacer les n disques supérieurs sur la tige U (pas forcément la tige à droite de T) et que la 3e tige soit nommée V. On ne suppose pas que les tiges U et V soient vides. Pour effectuer le déplacement, il suffit, comme dans le cas précédent, de suivre les étapes suivantes :

- étape 1 : déplacer les $n-1$ disques supérieurs depuis la tige T vers la tige V,
- étape 2 : déplacer le n-ème disque de la tige T (à partir du haut) au-dessus de la pile de la tige U,
- étape 3 : déplacer les $n-1$ disques supérieurs depuis la tige V vers la tige U.

Passons au code. Dans la situation ci-dessus, T est la **source**, U est le **destination** et V est la tige **intermédiaire**. L'algorithme est réalisé par une fonction `hanoi(n, src, dstn, interm)`. Les étapes 1 et 3 correspondent à des appels de la fonction `hanoi` mais avec `n` − `1` au lieu de `n` et

des tiges permutées. Bien sûr, l'algorithme ci-dessus suppose que $n \geq 1$. Sinon, c'est que $n = 0$ et donc la fonction ne fera rien. D'où le code suivant :

```python
def hanoi(n, src, dstn, interm):
    if n > 0:
        hanoi(n - 1, src, interm, dstn)
        print("src =", src, "\ndstn =", dstn, "\n----------")
        hanoi(n - 1, interm, dstn, src)


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

```
src = A
dstn = C
----------
src = A
dstn = B
----------
src = C
dstn = B
----------
src = A
dstn = C
----------
src = B
dstn = A
----------
src = B
dstn = C
----------
src = A
dstn = C
----------
```

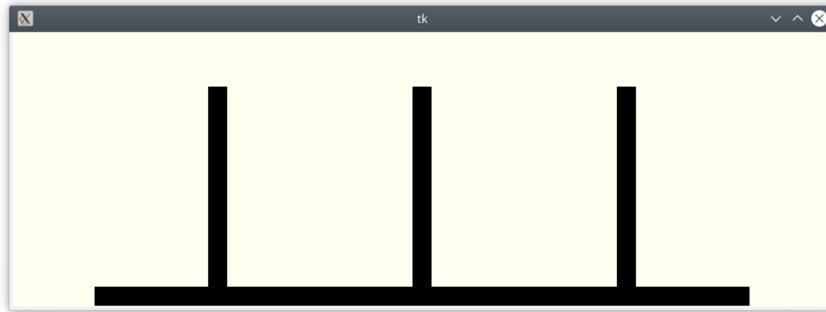
— Lignes 9-29 : on retrouve exactement les 7 déplacement décrits ci-dessus.
— Ligne 1 : selon la description précédant le code, on réalise `hanoi(n, T, U, V)`
— Ligne 3 : lorsqu'on déplace les $n - 1$ tiges de T vers la tige V, on fait l'appel `hanoi(n - 1, T, V, U)`
— Ligne 5 : lorsqu'on déplace les $n - 1$ tiges de V vers la tige U, on fait l'appel `hanoi(n - 1, V, U, T)`

Dessiner le décor en Tkinter

En Tkinter, les disques seront représentés par des rectangles rouges sur un canevas. Les ids des rectangles seront répertoriés dans une liste nommée `ids`. S'il y a initialement `n` disques, ils seront représentés dans l'ordre croissant des diamètres par `ids[0]`, `ids[1]`, etc.

Pour faciliter le codage du dessin, je choisirai un repère provisoire dont l'origine sera en bas à gauche du canevas et avec des axes orientés comme en maths. Pour passer des coordonnées `(x, y)` de ce repère aux coordonnées `(X, Y)` dans le repère du canevas de Tkinter, j'utiliserai une fonction `chgt(X, Y, center)`.

Dessinons le canevas, les trois tiges et le socle :

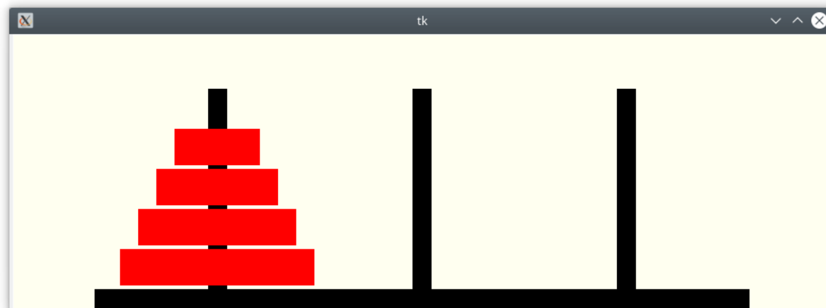



```
from tkinter import *

WIDTH_CNV = 900
HEIGHT_CNV = 300

# distance entre deux tiges
DIST = WIDTH_CNV / 4

# hauteur d'un disque
TOP = 4 * HEIGHT_CNV / 5

# disque le plus large
BASE = WIDTH_CNV / 4.2

# décalage de largeur entre deux disques
OFFSET = 20

# hauteur d'un disque
HEIGHT = 40
# espacement entre deux disques
SEP = 4

# dimensions du socle
SOCLE_LEFT = WIDTH_CNV / 10
SOCLE_RIGHT = 9 * WIDTH_CNV / 10
THICKNESS = 20


# origine du nouveau repère
ORIGN = (0, HEIGHT_CNV)


def chgt(X, Y, center=ORIGN):
    return (X + center[0], -Y + center[1])


root = Tk()
```

```python
cnv = Canvas(root, width=WIDTH_CNV, height=HEIGHT_CNV, bg="ivory")
cnv.pack()

# Le socle
A = chgt(SOCLE_LEFT, 0)
B = chgt(SOCLE_RIGHT, THICKNESS)
cnv.create_rectangle(A, B, fill="black")

# Les tiges
for i in range(3):
    x = (i + 1) * DIST - THICKNESS / 2
    A = chgt(x, 0)
    B = chgt(x + THICKNESS, TOP)
    cnv.create_rectangle(A, B, fill="black")

root.mainloop()
```

— Ligne 33 : pour alléger les appels (ligne 43 par exemple) de cette fonction, le paramètre `center` prend une valeur par défaut `ORIGN` (ligne 30).

Passons au placement des disques sur la tige de gauche :




```python
from tkinter import *

WIDTH_CNV = 900
HEIGHT_CNV = 300

# distance entre deux tiges
DIST = WIDTH_CNV / 4

# disque le plus large
BASE = WIDTH_CNV / 4.2

# décalage de largeur entre deux disques
OFFSET = 20

# hauteur d'un disque
HEIGHT = 40
```

```python
# espacement entre deux disques
SEP = 4

# dimensions du socle
SOCLE_LEFT = WIDTH_CNV / 10
SOCLE_RIGHT = 9 * WIDTH_CNV / 10
THICKNESS = 20
TOP = 4 * HEIGHT_CNV / 5

# origine du nouveau repère
ORIGN = (0, HEIGHT_CNV)

def chgt(X, Y, center):
    return (X + center[0], -Y + center[1])


root = Tk()

cnv = Canvas(root, width=WIDTH_CNV, height=HEIGHT_CNV, bg="ivory")
cnv.pack()

# Le socle
A = chgt(SOCLE_LEFT, 0, ORIGN)
B = chgt(SOCLE_RIGHT, THICKNESS, ORIGN)
cnv.create_rectangle(A, B, fill="black")

# Les tiges
for i in range(3):
    x = (i + 1) * DIST - THICKNESS / 2
    A = chgt(x, 0, ORIGN)
    B = chgt(x + THICKNESS, TOP, ORIGN)
    cnv.create_rectangle(A, B, fill="black")

# Les disques
n = 4
w = BASE
x = DIST - w / 2
y = HEIGHT / 2 + SEP

ids = []

for _ in range(n):
    A = chgt(x, y, ORIGN)
    B = chgt(x + w, y + HEIGHT, ORIGN)
    rect = cnv.create_rectangle(A, B, fill="red", outline="")
    ids.append(rect)
    x += OFFSET
    y += HEIGHT + SEP
```

```
65      w -= 2 * OFFSET
66
67 root.mainloop()
```

— Lignes 48 ou 61 : chaque disque est dessiné avec la méthode `create_rectangle`.

— Ligne 58 : on utilise une boucle `for` sur le nombre de disques ce qui permet de n'avoir à se soucier que du placement du disque le plus bas, les autres étant obtenus par décalage (lignes 63-65).

— Ligne 64 : les disques ne sont pas collés, il y a un espacement entre eux (`SEP`, ligne 18)

Adaptation de l'algorithme pour Tkinter

Reprenons notre algorithme récursif mais en numérotant les tiges 0, 1 et 2 :

```python
def hanoi(n, src, dstn, interm):
    if n > 0:
        hanoi(n - 1, src, interm, dstn)
        print("src =", src, "\ndstn =", dstn, "\n----------")
        hanoi(n - 1, interm, dstn, src)


hanoi(3, 0, 2, 1)
```

```
src = 0
dstn = 2
----------
src = 0
dstn = 1
----------
src = 2
dstn = 1
----------
src = 0
dstn = 2
----------
src = 1
dstn = 0
----------
src = 1
dstn = 2
----------
src = 0
dstn = 2
----------
```

Le problème est que cet algorithme n'est pas adapté à notre animation en Tkinter. En effet pour déplacer un item, nous utilisons la méthode `move` et nous devons connaître :

— l'id de l'item,

— le vecteur de déplacement.

Or, l'algorithme ci-dessus nous indique seulement quelles tiges (source et destination) sont concernées par un déplacement. Il nous faudrait donc au moins une référence vers l'item. Pour cela, et en vue d'adapter l'algorithme, on va numéroter les n disques, de bas en haut, par 0, 1, …, n − 1.

D'autre part , concernant le vecteur de déplacement, il ne dépend que des tiges de départ et d'arrivée et de la différence de hauteur dans les piles entre le départ et l'arrivée. Donc, on va modifier la fonction récursive en sorte qu'à chaque appel de la fonction `hanoi` soient donnés en arguments :

- le contenu de la pile source du déplacement,
- la liste des hauteurs de chacune des trois piles.

D'où le code suivant :

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


hanoi([0, 1, 2], 0, 2, 1, [3, 0, 0])
```

dont la sortie

```
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)
```

va être commentée en détail. On lit 7 lignes qui correspondent aux 7 déplacements. Chaque déplacement est décrit avec trois éléments, prenons par exemple le cas de la 2e ligne `1 (0, 1) (2, 0)` :

- le nombre au début indique le numéro de l'item qui est déplacé, ici 1 ;
- le couple qui suit est de la forme `(tige, etage)` ; il concerne la position du disque avant déplacement ; par exemple, `(0, 1)` signifie que le disque déplacé est initialement sur la tige 0 et à l'étage 1, les étages commençant à 0 ;
- le dernier couple concerne, lui aussi, la position du disque **après** déplacement ; par exemple, `(2, 0)` signifie que le disque est placé à la tige 2 et qu'il prend le niveau 0 (donc sur le socle).

Maintenant, commentons le code :

```
1 def hanoi(L, source, but, temp, hauteurs):
2     if L:
3         hanoi(L[1:], source, temp, but, hauteurs)
4         print(L[0], (source, hauteurs[source] - 1),
5               (but, hauteurs[but]))
```

```
6         hauteurs[source] -= 1
7         hauteurs[but] += 1
8         hanoi(L[1:], temp, but, source, hauteurs)
9
10
11 hanoi([0, 1, 2], 0, 2, 1, [3, 0, 0])
```

- Ligne 1 : `L` est la liste des numéros de disques de la pile que l'on va déplacer depuis la tige `source` vers la tige `but` ; le début de la liste indique le disque le plus bas. `L` n'est pas nécessairement la liste de **tous** les disques de la tige, mais uniquement les disques à déplacer.
- Ligne 2 : si la tige est vide, la fonction ne fait rien (et n'affiche rien en particulier).
- Ligne 3 : comme dans le premier algorithme, déplacement des `k` − 1 plus hauts éléments de la pile, en supposant que `k` soit le nombre d'éléments de `L`.
- Lignes 4-5 : le `k`^{e} disque est déplacé là où il doit aller.
- Lignes 6-7 : `hauteurs` désigne la liste des hauteurs des tiges 0, 1 et 2. La fonction `hanoi` doit conserver un état valide des hauteurs.

  Voilà pourquoi, suite à la ligne 3, hauteurs n'a pas été mis-à-jour.
- Lignes 6-7 : dès lors qu'on vient de déplacer le `k`^{e} disque, deux hauteurs de piles ont changé et il faut mettre à jour la liste `hauteurs`
- Ligne 8 : comme dans le premier algorithme, déplacement des `k` − 1 plus hauts éléments de la pile placés sur la tige initialement temporaire.

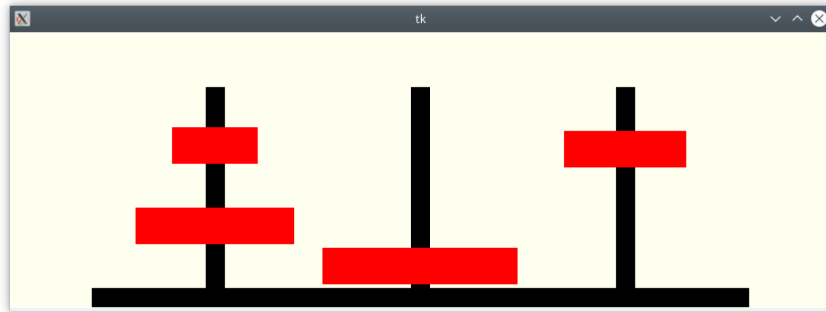
Déplacement d'un disque

Avant d'écrire l'animation, nous avons encore besoin de savoir comment déplacer un disque identifié par une id notée `nro` d'une position de départ `src =(tige0, etage0)` vers une destination `dstn =(tige1, etage1)`. Voici un code pour une telle fonction :

```
1 def move(nro, orig, dstn):
2     i, hi = orig
3     j, hj = dstn
4     dx = (j - i) * DIST
5     dy = -(hj - hi) * HEIGHT
6     cnv.move(ids[nro], dx, dy)
```

- Ligne 1 : `nro` est un indice d'item. On a vu dans le premier code que les id des disques sont stockées dans une liste `ids`. On pourra donc déplacer l'item souhaité (ligne 6).
- Ligne 1 : `orig` et `dstn` sont des couples `(tige, etage)`.
- Ligne 4 : les tiges étant régulièrement espacées, connaissant les indices des tiges, il est facile de calculer le vecteur horizontal de déplacement.
- Ligne 5 : les disques ayant tous la même épaisseur, c'est analogue pour le vecteur de déplacement vertical. Attention toutefois que l'axe des ordonnées du canevas étant orienté vers le bas et que les disques sont numérotés de manière croissante en se dirigeant vers le haut, il faut changer le signe de l'écart.

Ecrivons le code complet et testons-le :




```
from tkinter import *

WIDTH_CNV = 900
HEIGHT_CNV = 300

# distance entre deux tiges
DIST = WIDTH_CNV / 4

# disque le plus large
BASE = WIDTH_CNV / 4.2

# décalage de largeur entre deux disques
OFFSET = 20

# hauteur d'un disque
HEIGHT = 40
# espacement entre deux disques
SEP = 4

# dimensions du socle
SOCLE_LEFT = WIDTH_CNV / 10
SOCLE_RIGHT = 9 * WIDTH_CNV / 10
THICKNESS = 20
TOP = 4 * HEIGHT_CNV / 5

# origine du nouveau repère
ORIGN = (0, HEIGHT_CNV)

def chgt(X, Y, center):
    return (X + center[0], -Y + center[1])


root = Tk()

cnv = Canvas(root, width=WIDTH_CNV, height=HEIGHT_CNV, bg="ivory")
cnv.pack()

```

```python
# Le socle
A = chgt(SOCLE_LEFT, 0, ORIGN)
B = chgt(SOCLE_RIGHT, THICKNESS, ORIGN)
cnv.create_rectangle(A, B, fill="black")

# Les tiges
for i in range(3):
    x = (i + 1) * DIST - THICKNESS / 2
    A = chgt(x, 0, ORIGN)
    B = chgt(x + THICKNESS, TOP, ORIGN)
    cnv.create_rectangle(A, B, fill="black")

# Les disques
n = 4
w = BASE
x = DIST - w / 2
y = HEIGHT / 2 + SEP

ids = []

for _ in range(n):
    A = chgt(x, y, ORIGN)
    B = chgt(x + w, y + HEIGHT, ORIGN)
    rect = cnv.create_rectangle(A, B, fill="red", outline="")
    ids.append(rect)
    x += OFFSET
    y += HEIGHT + SEP
    w -= 2 * OFFSET


def move(nro, orig, dstn):
    i, hi = orig
    j, hj = dstn
    dx = (j - i) * DIST
    dy = -(hj - hi) * HEIGHT
    cnv.move(ids[nro], dx, dy)

move(2, (0,2), (2, 3))
move(0, (0,0), (1, 0))

root.mainloop()
```

— Ligne 75 : on déplace l'item 2 situé sur la tige 0 et à l'étage 2 sur la tige 2, à l'étage 3 (donc un niveau plus haut).

— Ligne 76 : de même, on déplace l'item 0 de la tige 0 et à l'étage 0 sur la tige 1, à l'étage 0 (donc au même niveau).

Animation pour n disques

On en arrive à la partie la plus délicate, le codage de l'animation. La difficulté vient du fait que l'algorithme récursif obtient de manière **instantanée** la suite de déplacements alors que l'animation est progressive. Le problème vient plus précisément du fait qu'on ne voit pas **comment faire pauser** la fonction récursive.

Ainsi, la manière la plus abordable semble de stocker la suite de déplacements et de demander à cette même fonction de dessiner les déplacements. C'est ce que fait le code ci-dessous :

```
1 def hanoi(L, source, but, temp, hauteurs, done):
2     if L:
3         A = hanoi(L[1:], source, temp, but, hauteurs, False)
4         B = [(L[0], (source, hauteurs[source] - 1), (but,
5                                                      hauteurs[but]))]
6         hauteurs[source] -= 1
7         hauteurs[but] += 1
8         C = hanoi(L[1:], temp, but, source, hauteurs, False)
9         if done :
10            moves = list(enumerate(A + B + C))
11            for (i, (nro, orig, dstn)) in moves:
12                cnv.after(1000 * (i + 1), move, nro, orig, dstn)
13        return A + B + C
14    return []
```

- Ligne 13 : la fonction `hanoi` renvoie la liste des déplacements.
- Lignes 11-12 : la fonction `hanoi` dessine un déplacement extrait de la liste des déplacements, cf. l'appel à `move` ligne 12 . Le déplacement a lieu toute les secondes; en effet, la méthode `after` est appelée après `1000 * (i + 1)` millisecondes donc toutes les `i + 1` secondes où `i` est l'indice du déplacement dans la liste `moves`.
- Les déplacements effectués suite à un appel de `hanoi` résultent du déplacement de la ligne 3, du déplacement du dernier disque (ligne 4) et le déplacement des disques (ligne 8) déjà changés de place. Le reste de l'algorithme est inchangé.

Voici le code complet :

```
from tkinter import *

WIDTH_CNV = 900
HEIGHT_CNV = 300

DIST = WIDTH_CNV / 4
BASE = WIDTH_CNV / 4.2
OFFSET = 20
HEIGHT = 40
SEP = 4
SOCLE_LEFT = WIDTH_CNV / 10
SOCLE_RIGHT = 9 * WIDTH_CNV / 10
THICKNESS = 20
TOP = 4 * HEIGHT_CNV / 5
ORIGN = (0, HEIGHT_CNV)
```

```python
def hanoi(L, source, but, temp, hauteurs, done):
    if L:
        A = hanoi(L[1:], source, temp, but, hauteurs, False)
        B = [(L[0], (source, hauteurs[source] - 1), (but,
                                                      hauteurs[but]))]
        hauteurs[source] -= 1
        hauteurs[but] += 1
        C = hanoi(L[1:], temp, but, source, hauteurs, False)
        if done :
            moves = list(enumerate(A + B + C))
            for (i, (nro, orig, dstn)) in moves:
                cnv.after(1000 * (i + 1), move, nro, orig, dstn)
        return A + B + C
    return []


def move(nro, orig, dstn):
    i, hi = orig
    j, hj = dstn
    dx = (j - i) * DIST
    dy = -(hj - hi) * HEIGHT
    cnv.move(ids[nro], dx, dy)


def chgt(X, Y, center):
    return (X + center[0], -Y + center[1])


root = Tk()

cnv = Canvas(root, width=WIDTH_CNV, height=HEIGHT_CNV, bg="ivory")
cnv.pack()

# Le socle
A = chgt(SOCLE_LEFT, 0, ORIGN)
B = chgt(SOCLE_RIGHT, THICKNESS, ORIGN)
cnv.create_rectangle(A, B, fill="black")

# Les tiges
for i in range(3):
    x = (i + 1) * DIST - THICKNESS / 2
    A = chgt(x, 0, ORIGN)
    B = chgt(x + THICKNESS, TOP, ORIGN)
    cnv.create_rectangle(A, B, fill="black")

# Les disques
n = 4
w = BASE
```

```python
x = DIST - w / 2
y = HEIGHT / 2 + SEP

ids = []

for _ in range(n):
    A = chgt(x, y, ORIGN)
    B = chgt(x + w, y + HEIGHT, ORIGN)
    rect = cnv.create_rectangle(A, B, fill="red", outline="")
    ids.append(rect)
    x += OFFSET
    y += HEIGHT + SEP
    w -= 2 * OFFSET


hanoi(list(range(n)), 0, 1, 2, [n, 0, 0], True)

root.mainloop()
```

Comment l'animation Tkinter est-elle réalisée dans le code de démonstration donné par CPython ? *Réponse* : l'animation est également codée par une fonction récursive mais sans stockage des déplacements. La fonction récursive est tout simplement mis en pause car **bloquée** dans une boucle while (lien vers les lignes du code-source) :

```python
1 while 1:
2     x1, y1, x2, y2 = c.bbox(p)
3     if y2 < ay1: break
4     c.move(p, 0, -1)
5     self.tk.update()
```

Comme on le voit ligne 4, le déplacement se fait par un écart de 1 pixel, sans que l'on puisse ajuster et l'animation est donc mal cadencée si la machine qui l'exécute est rapide.

Prolongements

Il est possible d'améliorer l'animation, comment on le voit avec cette application Javascript de mathisfun. Quelques suggestions :

— réécrire le code dans une classe Hanoi ;
— placer 4 boutons multimedia pour contrôler l'animation ;
— permettre à l'utilisateur de choisir le nombre de disques ;
— placer un compteur de déplacements ;
— donner à l'utilisateur la possibilité de déplacer lui-même les disques.