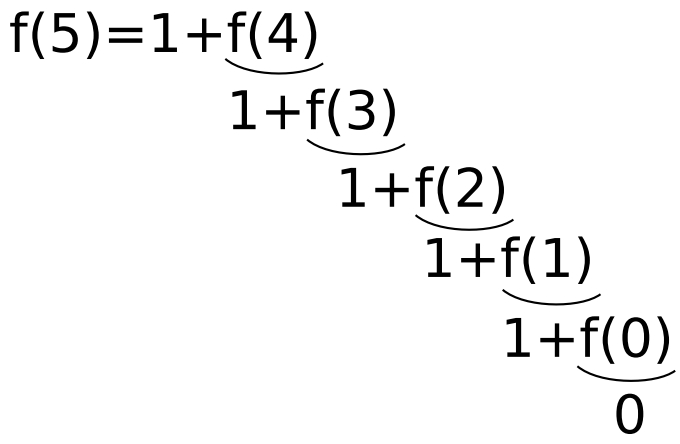
Récursivité

Récursivité

Fonction récursive = Fonction qui s'appelle elle même

```
def f(x):
    print(x)
    if x<=0:
        return 0
    else:
        return 1+f(x-1)

print(f(5))
```



Il faut une condition d'arrêt, sinon recursion infinie

```
def g(x):
    return g(x-1)
```

RuntimeError: maximum recursion depth exceeded

Factoriel

En itératif $n!=n\ (n-1)\ (n-2)\ \ldots\ 1$

```
def factoriel(n):
    valeur=1
    for x in range(1,n+1):
        valeur=valeur * x
    return valeur

print(factoriel(5))
```

Factoriel

En récursif $n!=n\,(n-1)\,(n-2)\,\ldots\,1$

$=n\,(n-1)!$

=> factoriel(n) = n x factoriel(n-1)

```
def factoriel(n):
    if n<=1:
        return 1
    else:
        return n*factoriel(n-1)
```

Combinaisons

$C^n_p = n!/((n-p)!\ p!)$

problèmes, chiffres très grands

$$C^n_p = C^{n-1}_p + C^{n-1}_{p-1}$$
$$C^n_0 = 1$$
$$C^n_p = 1$$

```
def combi(n,p):
  if n==p or p==0:
    return 1
  else:
    return combi(n-1,p)+combi(n-1,p-1)
```

Récursion double
recalcule intégralement
combi(n-1,p-1) pour combi(n-1,p)

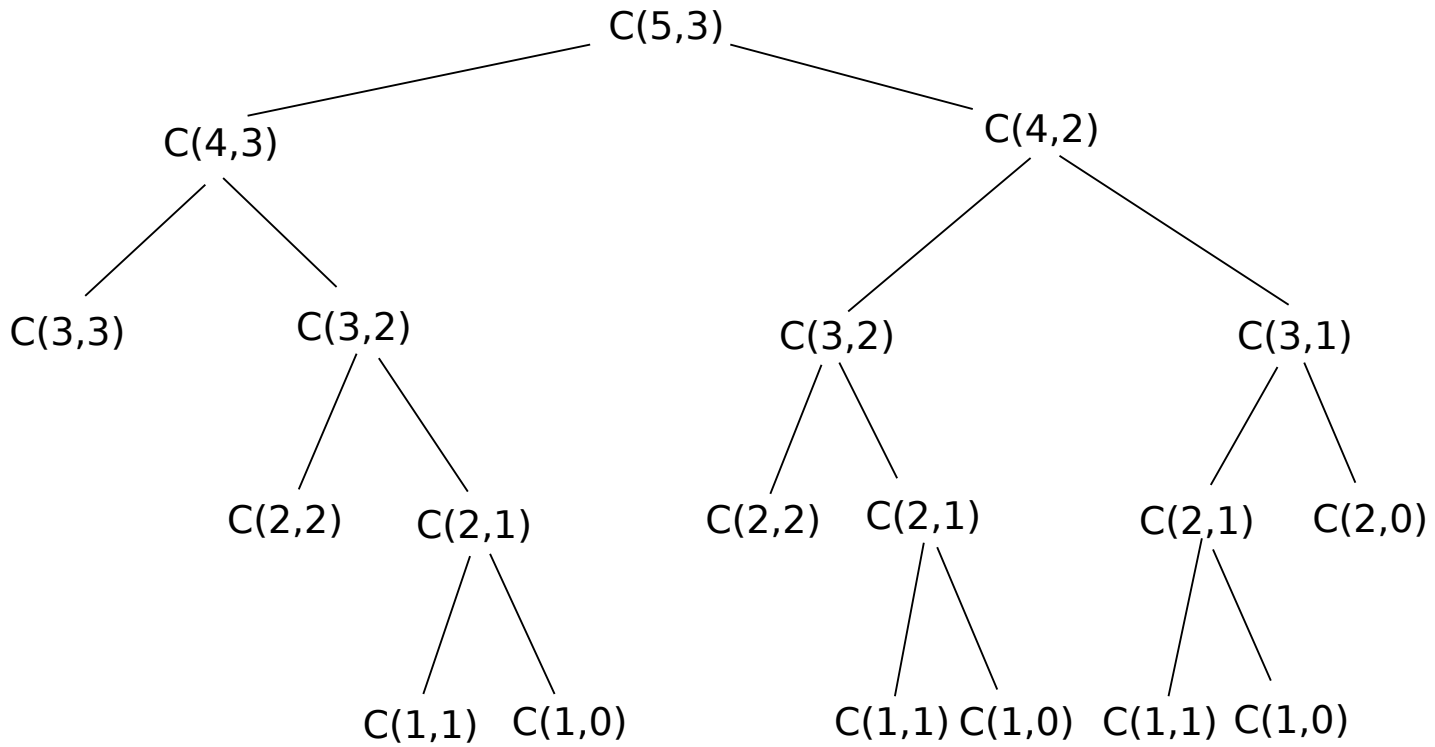
Combinaisons

```
def combi(n,p):
  if n==p or p==0:
    return 1
  else:
    return combi(n-1,p)+combi(n-1,p-1)
```

Pour chaque appel à combi(n,x)
=> 2 appels à combi(n-1,x) + On itère n fois

=> Complexité en $O(2^n)$

Combinaisons



C(25,13) demande 10 millions d'appels de fonctions

Combinaisons

On aurait pu remarquer:

$$C^{n}_{p} = (n/p)\ C^{n-1}_{p-1}$$

```
def combi2(n,p):
  if n==p or p==0:
    return 1
  else:
    return n/p*combi2(n-1,p-1)
```

1 seule récursion

division
(erreur d'arrondis)

Complexité en O(n)

Combinaisons

Coder les deux fonctions:

Comparer les temps de calculs pour n variant

```
def combi2(n,p):
  if n==p or p==0:
    return 1
  else:
    return n/p*combi2(n-1,p-1)
```

```
def combi(n,p):
  if n==p or p==0:
    return 1
  else:
    return combi(n-1,p)+combi(n-1,p-1)
```

```
import time

start=time.time()
[...]
end=time.time()

print(end-start)
```

Recursion

Avantages

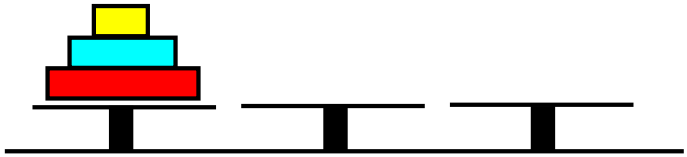
- Expression concise de certains problèmes

Inconvénients

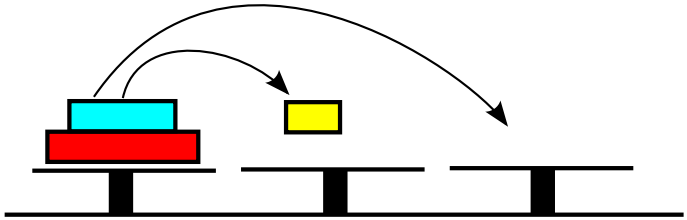
- Terminaison ?
- Complexité croit rapidement
- Utilisation de la pile

En développement:
A complexité égale, préférer une version itérative

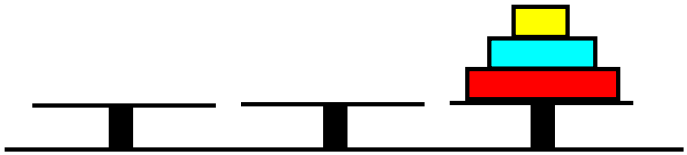
Tour de Hanoi



But:



...



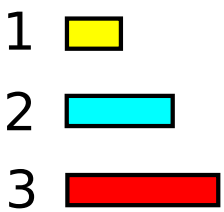
Contraintes:

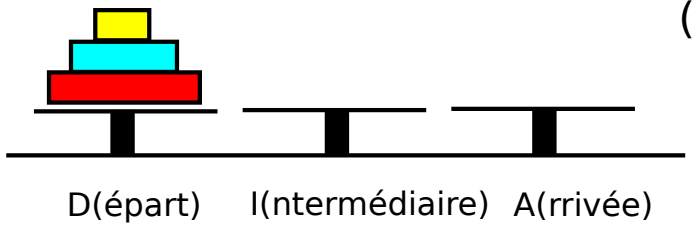
1 seul déplacement à la fois
(le jeton en haut de la pile)

le jeton doit se trouver sur un
jeton plus grand (ou sur le plateau)

Tour de Hanoi

Modèle





(n=3 jetons)

Algorithme:

```
Hanoi(n,D,A,I):
    Si n>0:
        Hanoi(n-1,D,I,A)
        D placer jeton de D vers A
        Hanoi(n-1,I,A,D)
```

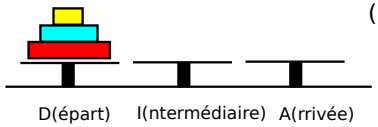
Implémenter un déplacement pour une tour de Hanoi de taille n

Combien de déplacement pour n=20 ?

Tour de Hanoi

Modèle

1
2
3



(n=3 jetons)

Algorithme:

```
Hanoi(n,D,A,I):
   Si n>0:
       Hanoi(n-1,D,I,A)
       D placer jeton de D vers A
       Hanoi(n-1,I,A,D)
```

Implémenter un déplacement pour une tour de Hanoi de taille n

Combien de déplacement pour n=20 ?

```
N=20

T=[[],[],[]]
T[0]=list(range(N,0,-1));

deplacement=[0]
def hanoi(n,D,A,I):
    if n!=0:
        hanoi(n-1,D,I,A)
        deplacement[0]+=1
        T[A].append(T[D][-1])
        T[D].pop()
        hanoi(n-1,I,A,D)


hanoi(N,0,2,1)
print(T)
print(deplacement)
```