

# Une approche de la complexité en classe.

Gilles Aldon - Jérôme Germoni - Jean-Manuel Mény

Journées nationales APMEP – Marseille 2013

# Fibonacci à un rang

## En première ou en terminale.

On programme une suite  $u$  et une suite  $v$  de la façon suivante :

```
def u(n):  
    if n==0 : return 1  
    return u(n-1)+u(n-1)  
  
def v(n):  
    if n==0 : return 1  
    return 2*v(n-1)
```

Calculs des premiers termes

# Questions

- 1 Comparer les sorties des deux programmes.
- 2 Ces deux programmes sont-ils « équivalents » ?

# Temps de calcul expérimental

Lancer les calculs de  $u(26)$  et  $v(26)$ .

Calculs de  $u(26)$  et  $v(26)$

Le couple (entrée , sortie) semble-t-il suffisant pour déclarer des programmes « équivalents » ?

# Ces deux programmes sont-ils « équivalents » ?

Mesures expérimentales des temps de calcul.

Courbes des temps de calcul

Explication pour les courbes observées ?

# Un exercice classique sur les suites : explication des temps de calcul

Notons  $s(n)$  le nombre d'opérations utilisées pour le calcul de  $v(n)$ .

```
def v(n):  
    if n==0 : return 1  
    return 2*v(n-1)
```

Expression de  $s(n)$  en fonction de  $n$  ?

# Un exercice classique sur les suites

$s(n)$  le nombre d'opérations utilisées pour le calcul de  $v(n)$ .

$s(0) = 0$  et  $s(n) = 1 + s(n - 1)$ . D'où  $s(n) = n$ .

# Un exercice classique sur les suites

Notons  $t(n)$  le nombre d'opérations utilisées pour le calcul de  $u(n)$ .

```
def u(n):  
    if n==0 : return 1  
    return u(n-1)+u(n-1)
```

Expression de  $t(n)$  en fonction de  $n$  ?



# Un exercice classique sur les suites

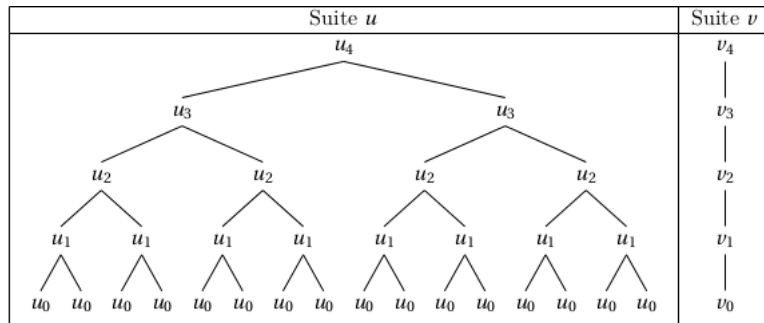
$t(n)$  le nombre d'opérations utilisées pour le calcul de  $u(n)$ .

$$t(0) = 0 \text{ et } t(n) = 1 + t(n-1) + t(n-1).$$

Suite arithmético-géométrique.

$$t(n) = 2^n - 1.$$

# Arbres des appels



# Questions complémentaires

Un test sur ma machine montre que  $u(22)$  est calculé en environ 1,26 s.

- ① En déduire une estimation du temps nécessaire au calcul de  $u(27)$ .  
Puis vérifier expérimentalement.
- ② Estimer alors le temps nécessaire au calcul de  $u(50)$ . ... Puis lancer le calcul de  $v(50)$

## Questions complémentaires : réponses

- ① Si l'on estime que le temps d'exécution est essentiellement dû aux opérations élémentaires (ici des additions) et qu'une addition prend un temps constant  $a$ , alors le calcul de  $u(22)$  nécessite un temps égal d'environ  $t_{22} = 2^{22}a$  et le calcul de  $u(27)$  demande un temps de calcul d'environ  $2^{27}a = 2^5 \times t_{22} \approx 40$  s. Confirmation expérimentale

## Questions complémentaires : réponses

- 1 Si l'on estime que le temps d'exécution est essentiellement dû aux opérations élémentaires (ici des additions) et qu'une addition prend un temps constant  $a$ , alors le calcul de  $u(22)$  nécessite un temps égal d'environ  $t_{22} = 2^{22}a$  et le calcul de  $u(27)$  demande un temps de calcul d'environ  $2^{27}a = 2^5 \times t_{22} \approx 40$  s. Confirmation expérimentale
- 2 Pour  $u(50)$  :  $1,26 \times 2^{50-22}$  secondes, soit plus de 10 ans.

## Questions complémentaires : réponses

- 1 Si l'on estime que le temps d'exécution est essentiellement dû aux opérations élémentaires (ici des additions) et qu'une addition prend un temps constant  $a$ , alors le calcul de  $u(22)$  nécessite un temps égal d'environ  $t_{22} = 2^{22}a$  et le calcul de  $u(27)$  demande un temps de calcul d'environ  $2^{27}a = 2^5 \times t_{22} \approx 40$  s. Confirmation expérimentale
- 2 Pour  $u(50)$  :  $1,26 \times 2^{50-22}$  secondes, soit plus de 10 ans.
- 3 Le calcul de  $v(50)$  est lui quasi immédiat.

# Un classique revisité

- ① On plie en deux une feuille de papier d'épaisseur 0,1 mm puis on recommence et on recommence. . . Duschmoll annonce qu'il est parvenu à la plier 30 fois sur elle-même. . .
- ② Après avoir constaté un temps de calcul de 1,26 s pour le calcul de  $u(22)$ , Duschmoll annonce que son programme lui donne aussi  $u(50) = 1\,125\,899\,906\,842\,624 \dots$

# Un exemple en classe de seconde

Triplets pythagoriciens de somme donnée.

On appelle triangle entier un triangle dont les côtés sont de longueur entière.



# Un exemple en classe de seconde

Avec la fonction suivante, on cherche à déterminer les triangles rectangles entiers de périmètre  $p$ .

```
def triangle_rect_de_perim(p):  
    L=[]  
    for a in range(1,p+1):  
        for b in range(1,p+1):  
            for c in range(1,p+1):  
                if a+b+c==p and a**2+b**2==c**2:  
                    L.append((a,b,c))  
    return L
```

Si on trace la courbe d'une fonction qui à  $p$  associe le temps d'exécution, à quoi peut-on s'attendre ?

# Triplets pythagoriciens de somme donnée

Obtenir les solutions avec un algorithme en temps quadratique.

# Triplets pythagoriciens de somme donnée

Obtenir les solutions avec un algorithme en temps quadratique.

```
def trp(p):  
    L=[]  
    for a in range(1,p+1):  
        for b in range(1,p+1):  
            c=p-a-b  
            if a**2+b**2==c**2:  
                L.append((a,b,c))  
    return L
```

Temps second degré

# Un exercice de comparaison

On propose ci-dessous deux fonctions python.

- 1 Expliquer pourquoi elles conviennent également pour la résolution du problème de la recherche des triangles rectangles entiers de périmètre  $p$ .
- 2 Quelle est la meilleure des deux ?

Les deux programmes

# Un exercice de comparaison

```
def trp1(p):  
    L=[]  
    for a in range(1,p/3+1):  
        for b in range(a,p/2+1):  
            c=p-a-b  
            if a**2+b**2==c**2:L.append((a,b,c))  
    return L
```

```
def trp2(p):  
    L=[]  
    for a in range(1,p/3+1):  
        for b in range(a,p-a):  
            c=p-a-b  
            if a**2+b**2==c**2:L.append((a,b,c))  
    return L
```

# Triplets pythagoriciens de somme donnée

- ① Pour  $1 \leq a \leq \frac{p}{3}$  et  $a \leq b \leq \frac{p}{2}$ , on a  $p - a - b > 0$ . De même pour  $1 \leq a \leq \frac{p}{3}$  et  $a \leq b \leq p - a$ .

Les deux fonctions sortiront donc des triplets  $(a, b, c)$  avec  $1 \leq a \leq b < c$  et  $a + b + c = p$ ,  $a^2 + b^2 = c^2$ .

- ② Pour tpr1.

Soit  $a$  tel que  $a > \frac{p}{3}$  avec  $a \leq b < c$  alors  $a + b + c > p$  : il n'existe pas de solution avec  $a > \frac{p}{3}$  ou avec  $b > \frac{p}{2}$ .

De même : pas de triplet  $(a; b; c)$  avec  $a \leq b < c$  et  $b > \frac{p}{2}$ .

idem pour tpr2.

- ③ La boucle en  $b$  s'arrêtant à  $\frac{p}{2}$  est meilleure puisque pour  $a \leq \frac{p}{3}$ , on a  $p - a \geq p - \frac{p}{3} > \frac{p}{2}$ .

# Le facteur sonne toujours trois fois.

## Un exemple en terminale.

Le facteur engage la conversation avec madame X à qui il vient de donner son courrier. La semaine dernière, madame X lui a dit avoir trois enfants et lui a demandé de deviner leurs âges. Avant-hier, elle lui a donné une indication en lui affirmant que le produit de leur âge est 36. Hier, elle lui donné une autre indication en ajoutant que la somme de leurs âges est égale au numéro de la maison. Le facteur affirme ne pas pouvoir encore conclure. Madame X affirme alors que son aînée est blonde. Le facteur connaît alors les âges des enfants.

# Le facteur sonne toujours trois fois.

Pour résoudre le problème, Tartempion propose le petit programme ci-dessous :

```
def TriFacteur(n):  
    L=[]  
    for a in range(n+1):  
        for b in range(n+1):  
            for c in range(n+1):  
                if a*b*c==n:  
                    L.append((a,b,c))  
    return L
```

si l'on trace la fonction qui à  $n$  associe le temps d'exécution de la fonction, à quoi doit-on s'attendre pour l'allure de la courbe obtenue ?

Le facteur



# Un facteur qui sonne moins souvent

Objectif : écrire, justifier et évaluer en temps une version telle que la suivante :

```
def TriFacteur(n):  
    L=[]  
    A=int(n**(1.0/3.0))  
    B=int(n**0.5)  
    for a in range(1,A+1):  
        for b in range(a,B+1):  
            if n%(a*b)==0:  
                c=n//(a*b)  
                if c>=b:  
                    L.append((a,b,c))  
  
    return L
```

Le fast facteur

# Image d'un nombre par un polynôme

## Un exemple en classe de première

On représente le polynôme  $P(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$  par la liste de ses coefficients  $a = [a_0, a_1, \dots, a_n]$ .

Rôle et comparaison des temps de calcul des programmes suivants :

# Image d'un nombre par un polynôme

Rôle et comparaison des temps de calcul des programmes suivants :

```
def evaluate(a,x):  
    s=0  
    for i,ai in enumerate(a):  
        y=1  
        for j in range(i):  
            y*=x  
        s+=ai*y  
    return s
```

```
def eval(a,x):  
    s=0  
    y=1  
    for i,ai in enumerate(a):  
        s+=ai*y  
        y*=x  
  
    return s
```

- 1 La première fonction nécessite  $(i - 1) + 1 = i$  opérations (additions ou multiplications) au  $i$  ème passage dans la boucle. Soit  $1 + 2 + 3 + \dots + n = \frac{1}{2}n(n + 1)$  opérations, d'où la fonction du second degré observée.
- 2 La seconde nécessite 3 opérations à chaque exécution de la boucle, soit un total de  $3n$  opérations, d'où la fonction affine obtenue.

vérif expérimentale

# Autres exemples

Des algorithmes de référence accessibles au lycée :

- Hörner
- exponentiation rapide (  $x^n = (x^2)^{\frac{n}{2}} \dots$  )
- ...

# Et en ISN

Objectif :  
faire comprendre en un temps très court l'essentiel du problème de la complexité.

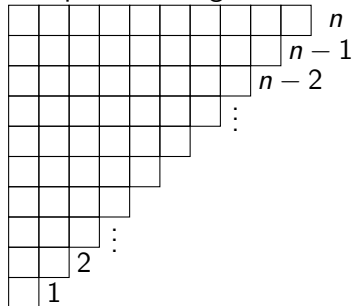
# Un algorithme qui ne fait rien...mais qui le fait en temps quadratique

Que dire du temps d'exécution de :

```
def triangle(n) :  
    for i in range(n) :  
        for j in range(i,n) :  
            pass
```

# Le triangle.

Complexité triangulaire :



$$\sum_{i=0}^{n-1} (n - i) = \frac{1}{2}n(n + 1).$$

Temps d'exécution proportionnel à  $\frac{1}{2}n(n + 1)$ .

Contrôle expérimental



# Un algo un peu moins inutile, svp

**Exercice.** Écrire un algorithme de tri d'une liste dont la complexité relève du schéma triangulaire précédent.

Exemple du tri par sélection (algorithme au programme de la spécialité ISN)

Tri par sélection

# Un algo un peu moins inutile, svp

Principe du tri par sélection d'une liste  $L[1], L[2], \dots, L[n]$  :

Pour  $j$  de 1 à  $n-1$  :

repérer l'élément minimum dans  $L[j+1], L[j+2], \dots, L[n]$   
échanger  $L[j]$  avec cet élément minimum

# Un algorithme qui ne fait rien...mais plus vite !

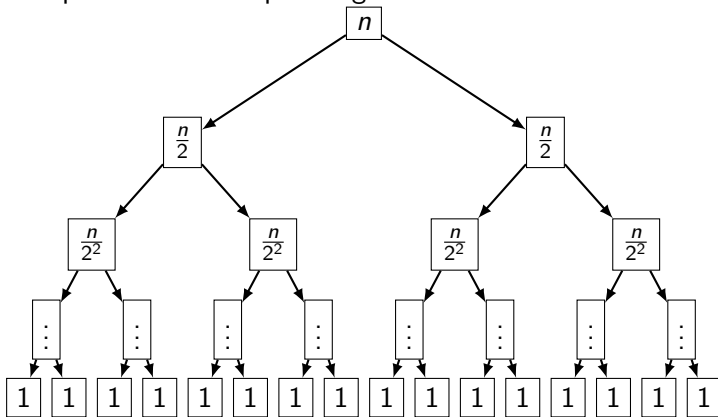
Que dire du temps d'exécution de :

```
def arbre(n) :  
    h,t=1,n  
    while t>=1 :  
        for j in range(h):  
            for k in range(t):  
                pass  
  
        h*=2  
        t/=2
```

contrôle expérimental

# Un algorithme qui ne fait rien...mais plus vite !

Complexité « diviser pour régner ».



# Arbre.

$$1 \times n + 2 \times \frac{n}{2} + 2^2 \times \frac{n}{2^2} + \cdots + 2^k \times \frac{n}{2^k} = k \times n \text{ avec } 2^k = n$$

$$1 \times n + 2 \times \frac{n}{2} + 2^2 \times \frac{n}{2^2} + \cdots + 2^k \times \frac{n}{2^k} = \log_2(n) \times n$$

Temps d'exécution proportionnel à  $n \log_2(n)$ .

contrôle expérimental

# Un algo un peu moins inutile, svp

**Exercice.** Écrire un algorithme de tri d'une liste dont la complexité relève du schéma précédent.

Exemple du tri par fusion (algorithme au programme de la spécialité ISN).

# Un algo un peu moins inutile, svp

## Principe du tri par fusion.

Si la liste contient au plus un élément, elle est triée. Sinon :

- 1 On la partage en deux listes gauche et droite de longueur à peu près égale.
- 2 On trie la sous-liste gauche et la sous-liste droite ainsi obtenues.
- 3 On fusionne ces deux sous-listes.

une traduction python