Introduction à l'algorithmique

Chargée de cours: **Lélia Blin**

Email: *lelia.blin@lip6.fr*

Transparents : http://www-npa.lip6.fr/~blin/Enseignements.html

Equation de base

Programme = Structures de données + algorithmes

Niklaus Wirth (concepteur du langage Pascal)

Le problème est de concevoir les structures de données de manière à ce que :

- La **mémoire** utilisée soit **minimale**
- Les **algorithmes** soient les plus **simples** et les plus **efficaces** possible

Structures de données

Ordinateur = Système de Traitement de l'Information

- Unité de base = le bit (0/1)
- 1 octet = 8 bits

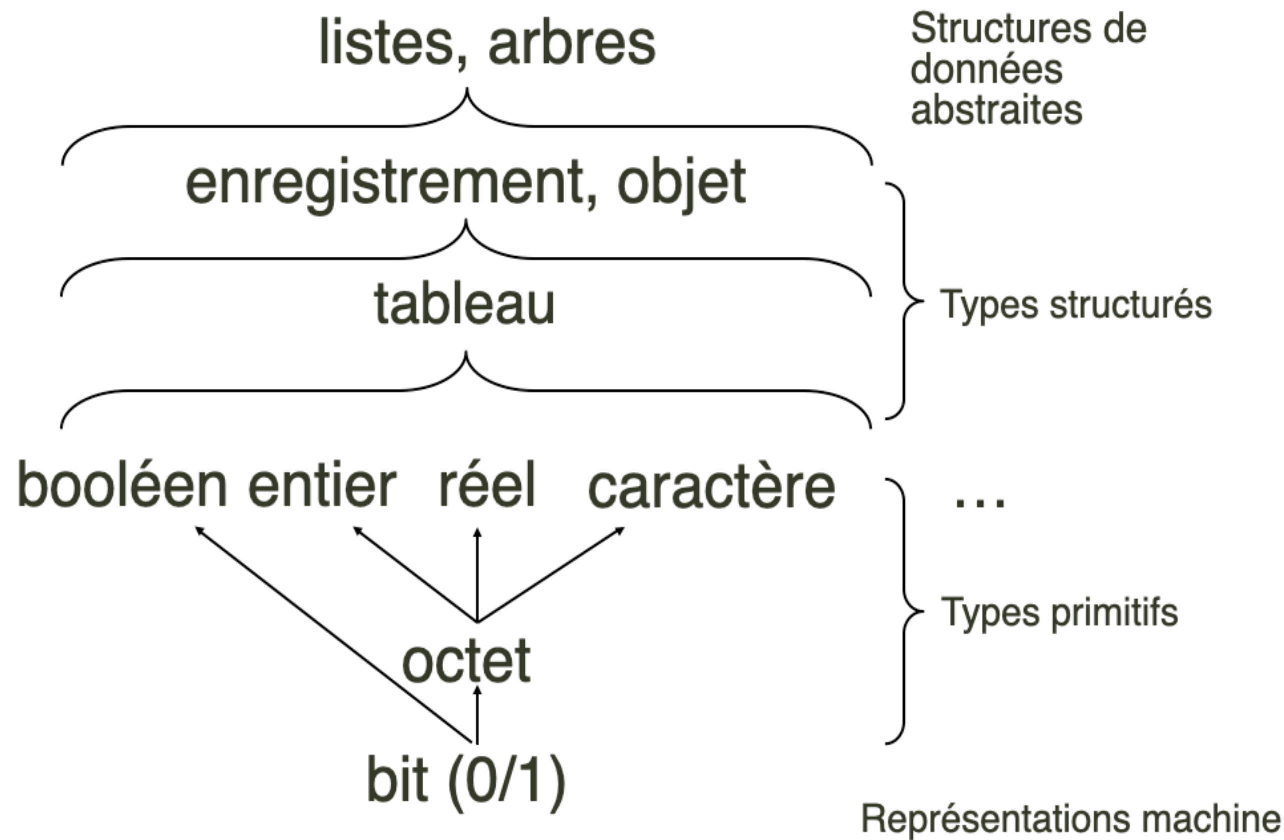
Problème:

- l'information n'est pas manipulable au même niveau par l'homme et par la machine

Solutions:

- Agrégation et structuration de l'information
- Construction de structures plus abstraites

Hiérarchie des données



Définition

- Un ensemble structuré de données est construite par agrégation et hiérarchisation de données ou structures de données plus primitives

- Définition récursive

Origine du terme Algorithmique

Abu Ja Far Mohamed Ibn Al-rhowâ-rismî

- Mathématicien, géographe, astrologue et astronome perse né vers 783 mort vers 850
- Il est à l'origine des mots:
    - Algorithmique (latinisé)
    - Algèbre (dont il est considéré comme le père)
- Il est à l'origine de l'utilisation des chiffres arabes

Notion d'algorithme

Un **algorithme** est la composition d'un ensemble fini d'étapes dont chacune est:

- définie de façon rigoureuse,
- définie de façon non ambiguë,
- effective (pouvant être réalisée en un temps fini)

Exemple d'algorithme

Calcul de PGCD entre 2 entiers n et m

Algorithme

```
def PGCD(n,m):
    r = n
    print("n,m,r")
    while(r!=0):
        print(n,m,r)
        r = n % m
        if(r!=0):
            n = m
            m = rR
    print("Resultat : ",m)
```

Exemple

```
entrer n
10
entrer m
35
n,m,r
10 35 10
35 10 10
10 5 5
Resultat :  5
```

Simplicité & Efficacité

Notion d'algorithme plus générale que celle de programme

Deux besoins (potentiellement contradictoires) :

Simplicité de l'algorithme

- A comprendre
- A mettre en œuvre
- A mettre au point

Efficacité de l'algorithme

- Place mémoire
- Vitesse d'exécution

Somme des entier de 1 à n

Algorithme naif

```
def somme(n):
    som=0
    for i in range(1,n+1):
        som = som + i
        print(i,':',som-i,"+",i,'=',som)
    return(som)

print("entrer un entier: ")
n=input()
print("resultat: ",somme(int(n)))
```

Exemple

```
entrer un entier:
7
1 : 0 + 1 = 1
2 : 1 + 2 = 3
3 : 3 + 3 = 6
4 : 6 + 4 = 10
5 : 10 + 5 = 15
6 : 15 + 6 = 21
7 : 21 + 7 = 28
resultat:  28
```

Questions

- Cet algorithme est-il efficace?
- Combien d'addition en fonction de n va faire cet algorithme?

Somme des entier de 1 à n

Critères mathématiques

$1 + 2 + 3 + \cdots + n$

$+$

$n + n - 1 + \cdots + 1$

$= (n+1) + (n+1) + \cdots + (n+1)$

$= n(n+1)$

Donc

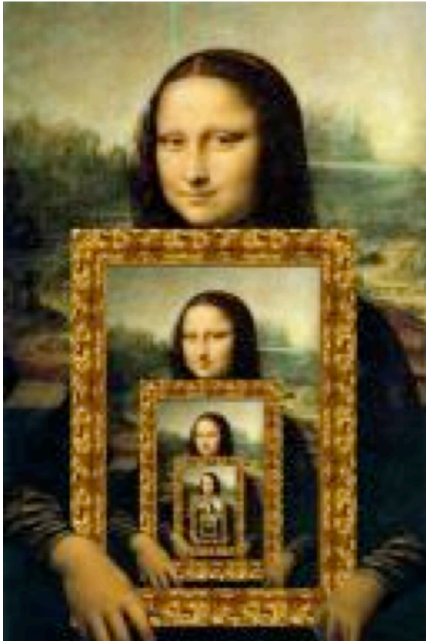
$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

Coût: 1 addition, 1 multiplication et 1 division (3 opérations)

Itératif ou récursif

Définition récursivité

Un algorithme récursif est un algorithme qui s'appelle lui même

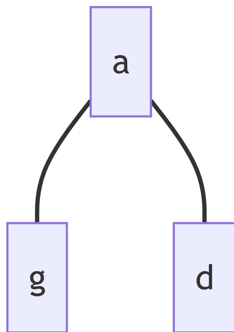


On oppose généralement les algorithmes récursifs aux algorithmes dits itératifs qui s'exécutent sans invoquer ou appeler explicitement l'algorithme lui-même.

Suite de fibonacci

Algorithme récursif

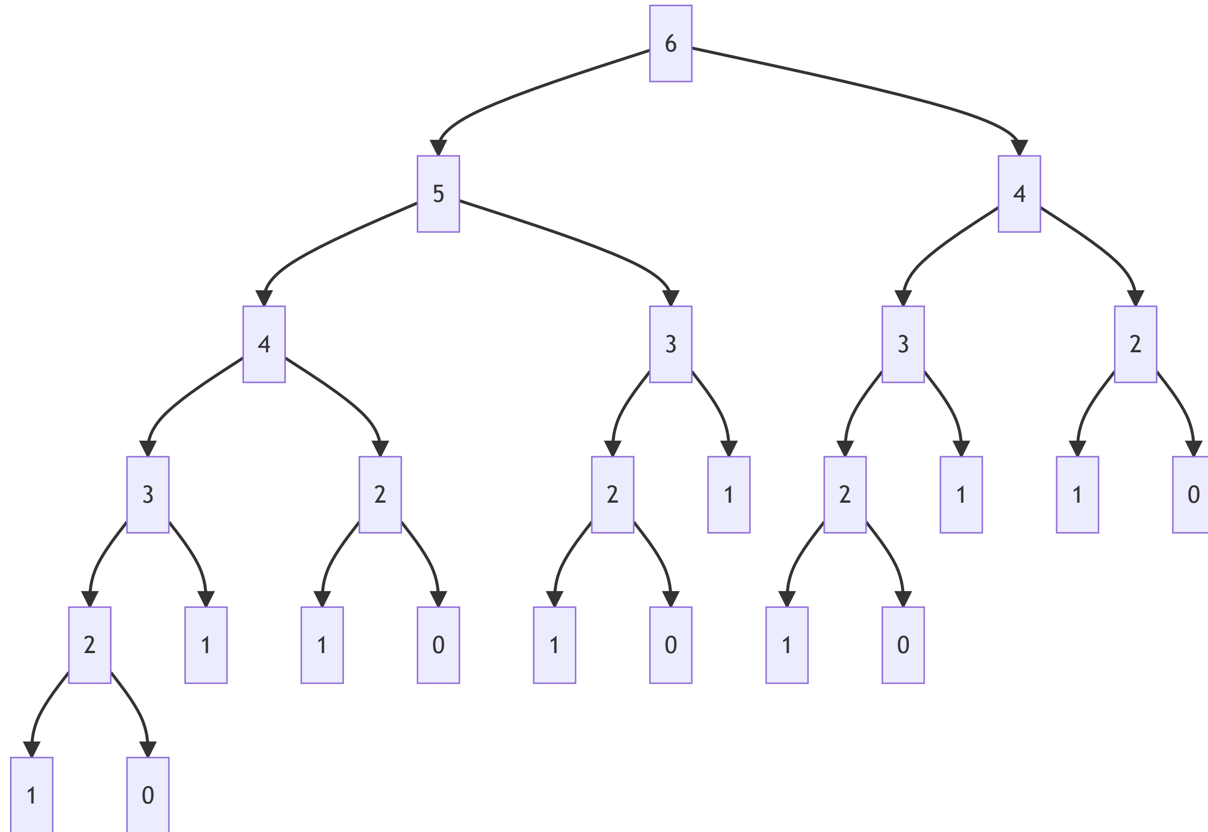
```
def fiboR(n,t):
    print("fibo(",n,") ->",t)
    if(n==0 or n==1):
        return(1)
    else:
        return(fiboR(n-1,t+"g")+fiboR(n-2,t+"d"))
```



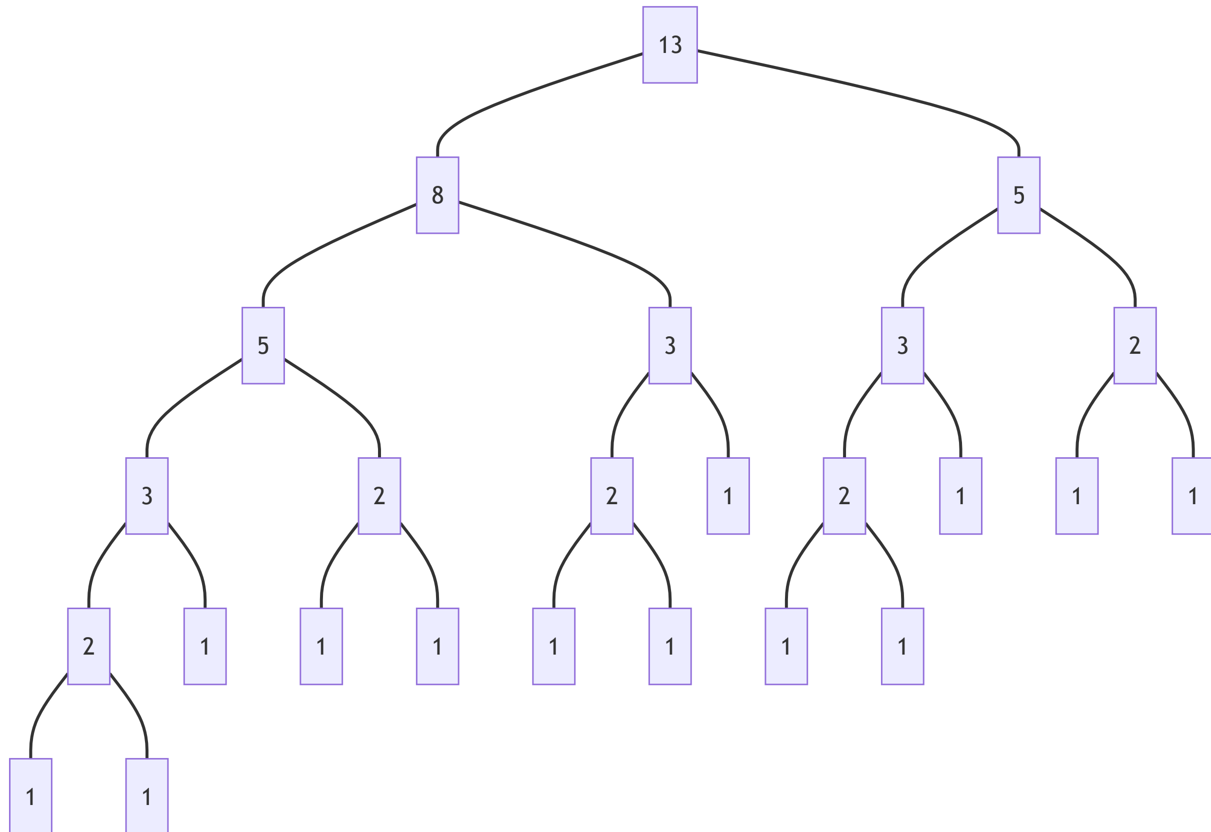
- Combien il y a t'il d'appel récurssif pour:
  - fibo(6)?
  - fibo(n)?

```
fibo( 6 ) -> a
fibo( 5 ) -> ag
fibo( 4 ) -> agg
fibo( 3 ) -> aggg
fibo( 2 ) -> agggg
fibo( 1 ) -> aggggg
fibo( 0 ) -> aggggd
fibo( 1 ) -> agggd
fibo( 2 ) -> aggd
fibo( 1 ) -> aggdg
fibo( 0 ) -> aggdd
fibo( 3 ) -> agd
fibo( 2 ) -> agdg
fibo( 1 ) -> agdgg
fibo( 0 ) -> agdgd
fibo( 1 ) -> agdd
fibo( 4 ) -> ad
fibo( 3 ) -> adg
fibo( 2 ) -> adgg
fibo( 1 ) -> adggg
fibo( 0 ) -> adggd
fibo( 1 ) -> adgd
fibo( 2 ) -> add
fibo( 1 ) -> addg
fibo( 0 ) -> addd
13
```

Arbre d'appel récursif [exemple fibo(6)]



Arbre d'appel récursif [exemple fibo(6)]



Suite de Fibonacci recursive

- Approche récursive: Simple
- L’approche récursive est-elle efficace?
- Quel est le nombre d’appels récursifs en fonction de l'éntrée n?

Suite de Fibonacci recursive

- Approche récursive: Simple
- L'approche récursive est-elle efficace?
- Quel est le nombre d'appels récursifs en fonction de l'éntrée n?
  - 2^n appels
  - Méthode très redondante
  - Coûteux voire rédhibitoire

Suite de Fibonacci itératif

Algorithme itératif avec tableau

```
def fiboIT(n):
    fib = [0 for p in range(0, n+1)]
    fib[0]=1;fib[1]=1
    for i in range(2,n+1):
        fib[i]=fib[i-1]+fib[i-2]
    return(fib)
```

Resultat

```
[1, 1, 2, 3, 5, 8, 13]
```

Algorithme itératif sans tableau

```
def fiboIsT(n):
    a=1;b=1
    r=0
    for i in range(2,n+1):
        r=a+b
        b=a
        a=r
    return(r)
```

Resultat

```
13
```

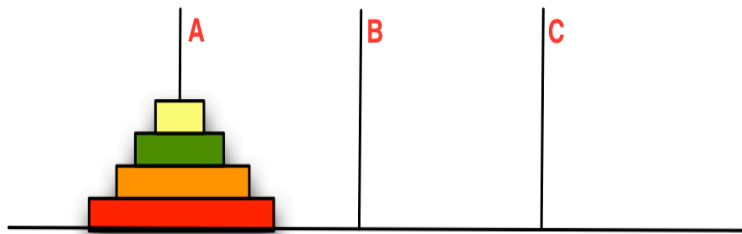
Les tours de Hanoï

Légende:

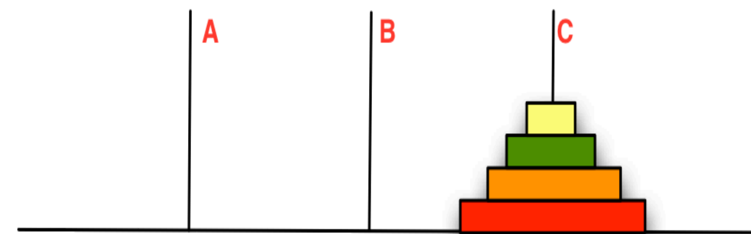
- Dans un temple dieu a installé trois aiguilles de diamant.
- Dieu enfila au commencement des siècles, 64 disques d'or pur, sur une des aiguilles, du plus gros au plus petit.
- Nuit et jour, les prêtres se succèdent sur les marches de l'autel, occupés à transporter les disques de la première aiguille sur la troisième.
- Ils doivent respecter deux règles:
  - On ne peut déplacer qu'un disque à la fois.
  - Un disque ne peut pas être disposé sur une disque plus petit.
- Quand tout sera fini, la troisième tour tombera et ce sera la fin du monde!
- Quand est prévu la fin du monde?

Jeu de réflexion imaginé par le mathématicien français Édouard Lucas

Les tours de Hanoï



Tour initiale



Tour finale

Quelle stratégie de jeu adopter?

Les tours de Hanoï

Stratégie

- déplacer la tour des $n-1$ premiers disques de A vers B.
- déplacer le plus grand disque de A vers C.
- déplacer la tour des $n-1$ premiers disques de B vers C.

Jouons un peu

[https://codepen.io/finnhvman/pen/gzmMaa?editors=1111]

Les tours de Hanoï : Algorithme récursif

```python
def hanoi(n,a="A",b="B",c="C"):
  if (n > 0):
      hanoi(n-1,a,c,b)
      print("de",a,"vers",c)
      hanoi(n-1,b,a,c)

print("Déplace un anneau")
print(hanoi(3))
```

```
Déplace un anneau
de A vers C
de A vers B
de C vers B
de A vers C
de B vers A
de B vers C
de A vers C
```

Les tours de Hanoï : approche récursive

Avantages : Simplicité

Nombres d'appels récursifs en fonction de n ?

Les tours de Hanoï : approche récursive

Avantages: Simplicité

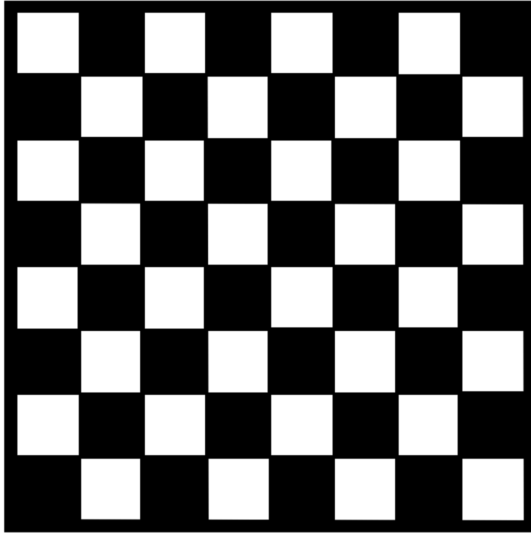
Nombres d'appels récursifs : $2^n - 1$

- Pour jeu à 64 disques requiert un minimum de 2^{64}-1 déplacements
- Soit 18 446 744 073 709 551 615 déplacements.
- En admettant qu'il faille une seconde pour déplacer un disque
- Soit 86 400 déplacements par jour.
- La fin du jeu aurait lieu au bout d'environ 213 000 milliards de jours
- Ce qui équivaut à peu près à 584,5 milliards d'années.
- Soit 43 fois l'âge estimé de l'univers (13,7 milliards d'années selon certaines sources)

Explosion combinatoire

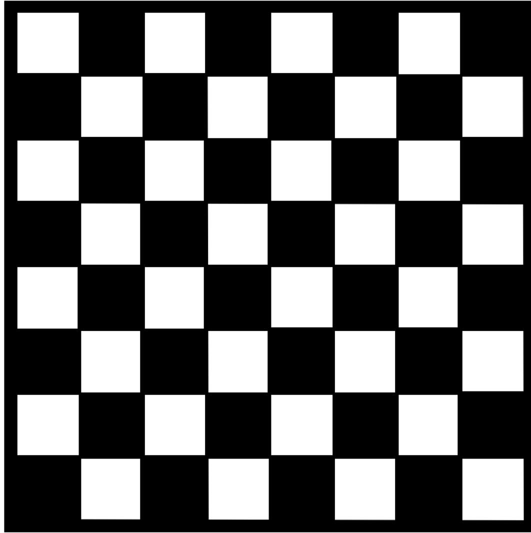
Les petits ruisseaux font les grandes rivières

Histoire de l'empereur

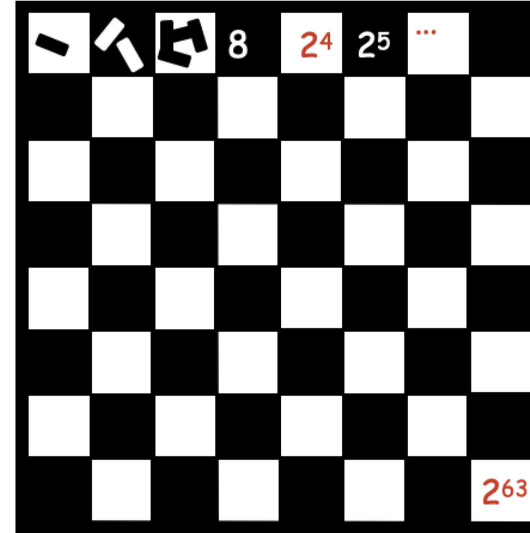


Echiquier

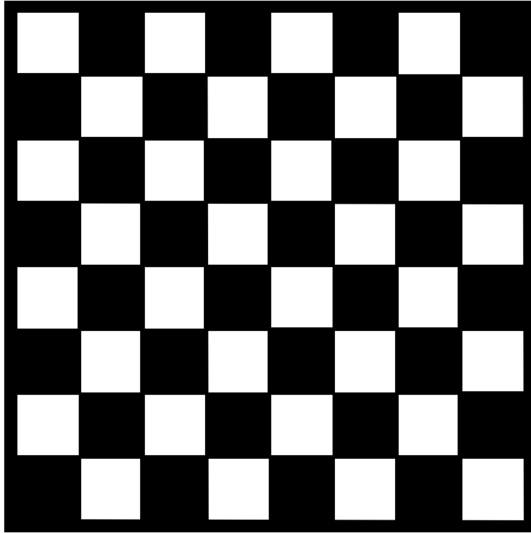
Histoire de l'empereur



Echiquier



Histoire de l'empereur



Echiquier

Combien de grains de riz sur l'échiquier

- Somme des grains de riz de chaque case
- Chaque case i a 2^i grains de riz
- $\sum_i 2^i = 2^{i+1} - 1 = 2^{64} - 1 \approx 16.10^{18}$ grains

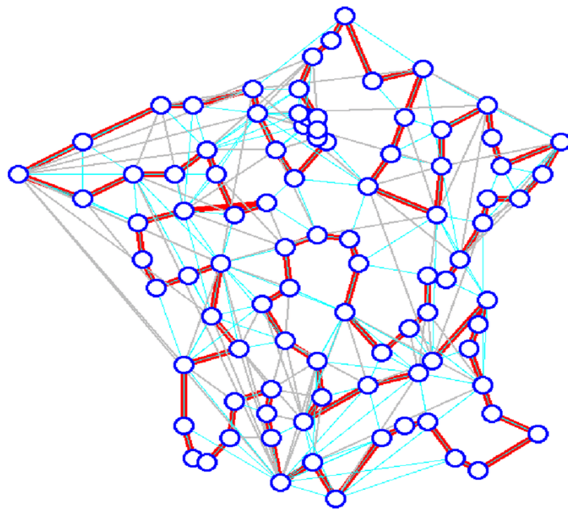
Grains de riz en poids

- 1000 grains de riz font environs 30 kg
- 16.10^{18} font environs 480.10^{15} kg
- soit environ 480.10^{12} Tonne

Combien d'année pour satisfaire le paysan?

- Production mondiale de riz par an
- 2009 record: 690.10^6 T de paddy
- Il faudra environ 700.000 ans

Le voyageur de commerce

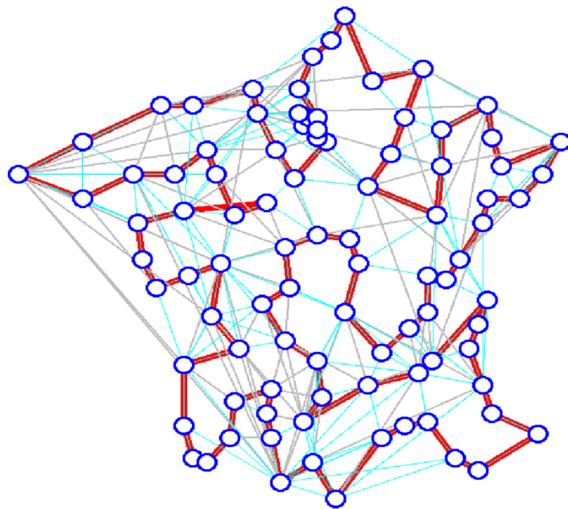


source de l'image

Problème:

- Soit n villes
- Trouver le circuit le plus court en nombre de kilomètres qui passe une est une seule fois par chaque ville.

Le voyageur de commerce



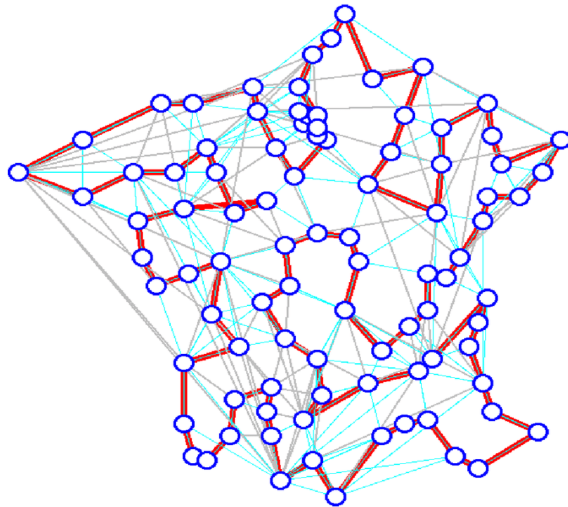
source de l'image

Solution naïve:

- Tester toutes les combinaisons possibles
- Calculer le coût de chaque combinaison
- Conserver la meilleure

Coût d'une telle approche?

Le voyageur de commerce



source de l'image

Coût d'une telle approche?

- Nombre de circuits : $(n-1)!$
- Nombre d'additions pour chaque circuit : $n-1$
- Nombre total d'additions :
  - $(n-1)*(n-1)!$

Calcul de la solution pour la France

Coût d'une telle approche?

- Nombre total d'additions : $(n-1)*(n-1)!$

Principales villes françaises

- En 2022, on compte 279 villes de plus de 30.000 habitants (wikipédia)
- Supposons qu'il y en ait que 30 villes
- il faut 265.10^{34} opérations

Temps de calcul pour 30 villes

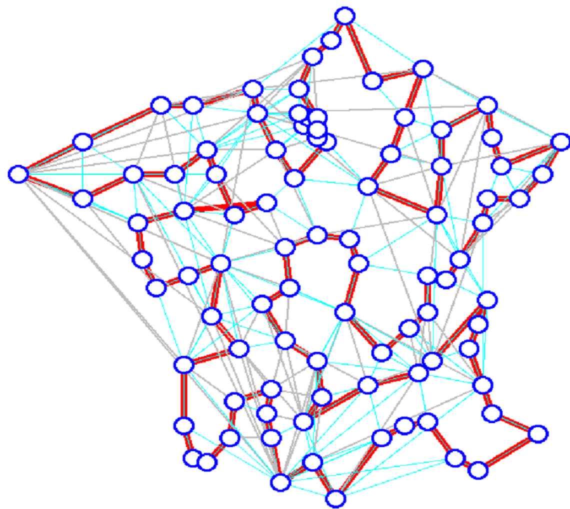
- En 2022 processeur le plus rapide: 1,1 exaFLOPS (wikipédia)
- soit $1,1.10^{18}$ opérations/seconde
- $265.10^{34}/1,1.10^{18} = 241.10^{16}s = 76.420.598.681$ années
- Univers: 15 milliards d'années
- $5,09$ l'âge de l'univers

Rapidité des ordinateurs

Temps de calcul pour 30 villes

- En 2022 processeur le plus rapide: 1,1 exaFLOPS (wikipédia)
  - soit $1,1.10^{18}$ opérations/seconde
  - $265.10^{34}/1,1.10^{18} = 241.10^{16}s = 76.420.598.681$ années
  - Univers: 15 milliards d'années
  - $5,09$ l'âge de l'univers
- juin 2011 processeur le plus rapide 8,162 PétaFLOPS
  - soit $8,162.10^{15}$ opérations/seconde
  - 3,24.1018s = 102.739.726.027 années
  - Univers: 15 milliards d'années
  - 6,8 l'âge de l'univers

Le voyageur de commerce: Conclusion



source de l'image

- Exemple d'algorithme correct mais inutilisable.
- Il n'existe pas de solution exacte en temps raisonnable.

Vocabulaires utilisée

- Un algorithme est une suite finie et non ambiguë d'instructions et d'opérations permettant de résoudre un classe de problème.
- Une heuristique est une méthode de calcul qui fournit rapidement une solution réalisable, pas nécessairement optimale ou exacte, pour un problème d'optimisation difficile.
- Un algorithme d'approximation est une méthode permettant de calculer une solution approchée à un problème d'optimisation.
- Un programme un algorithme et des structures de données dans un langage de programmation donné.

Etude de la complexitées

Etude de la complexité : Objectif

- Se donner des éléments théoriques d'appréciation, a priori, des performances des algorithmes

- Définir un ordre de grandeur
  - du temps d'exécution et
  - de l'espace mémoire utilisé par un algorithme
  - en fonction du nombre n d'éléments traités

Complexités

Complexité temporelle :

temps d'exécution de l'algorithme en fonction de l'entrée n

Complexité spatiale:

mémoire utilisé par un algorithme en fonction de l'entrée n

A quoi sert l'étude de la complexité

But:

- Se donner des éléments théoriques d'appréciation, a priori, des performances des algorithmes

Pratique :

- S'assurer que les performances de l'algorithme sont compatibles avec les performances des machines utilisés

Théorique :

- Comparer des algorithmes entre eux pour des problèmes de grande taille

Théorie de la complexité

Théorique :

- Comparer des algorithmes entre eux pour des problèmes de grande taille

Formuler les énoncés de la forme:

- Sur toute machine, et quel que soit le langage utilisé, l'algorithme A est meilleur que l'algorithme B pour des données de grande taille.

Théorie de la complexité : Etude

- Etudier la complexité d'un algorithme, c'est rechercher une fonction de n dont on est certain qu'elle majore la durée d'exécution, lorsque n devient très grand
  - le meilleurs des cas d'exécution
  - le pire des cas d'exécution
  - le temps moyen d'exécution

Complexité : Calcul

- Temps d'accès aux variables (cases mémoires)
  - en lecture/écriture : t_a
- Temps de calcul
  - opérations arithmétiques : t_{op}
  - comparaisons : t_c
- Exemple : c = a + b
  - accès à a et b en lecture, à c en écriture : $3t_a$
  - addition: t_{op}
  - total: $3t*a + t*{op}$

Complexité : Approche générale

- Choix des opérations à comptabiliser
- Comptage du nombre d'opérations exécutées

Complexité : Opération fondamentale

- L'opération fondamentale est celle telle que pour tout algorithme résolvant le problème, la complexité en sera une grandeur proportionnelle
- Exemples
  - Recherche d'un élément dans un tableau $\rightarrow$ comparaison
  - Recherche d'un élément sur un disque $\rightarrow$ accès disque
  - Tri d'un ensemble d'éléments $\rightarrow$ comparaison + déplacement d'élément

Complexité : Calcul

- Pour calculer la complexité temporelle il faut
  - évaluer le nombre de fois où l'opération fondamentale est exécutée par l'algorithme
  - Examiner le code de l'algorithme
- Le calcul peut varier en fonction des données données en entrée à l'algorithme

Séquences et alternatives

- Soit X un morceau de programme
- Soit $P(X)$ le nb d'opérations correspondant à l'exécution de X
- Séquence : $X_1; X_2$
  - $P(X_1; X_2) = P(X_1) + P(X_2)$
- Alternative : si C alors X_1 sinon X_2
  - On ne sait pas a priori quel bloc d'instructions va être exécuté -$P(C) + max(P(X_1), P(X_2))$

Boucles

- Il faut prendre en compte le nombre d'opérations $P(i)$ effectuées à itération i de la boucle
- $P(X) = \sum_i P(i)$
- Pb = on ne connaît pas forcément le nb total d'itérations
- Boucles for : on connaît n
- Boucles tant que ou répéter jusqu'à
  - Calculer une borne supérieure à partir de l'incrément
  - Recherche d'un majorant

Boucles FOR : exemple 1

```
def BoucleFor1(n):
    l1. r=1
    l2. for i in range(n):
    l3.     r=r*(i+1)
    l4. print(r)
```

- Total= $l1 + n(l2 + l3) + l4 = an + b \approx an$
- Complexité temporelle : $O(n)$

Boucles FOR : exemple 2

```
def BoucleFor2(n,m):
    l1. r=1
    l2. for i in range(n):
    l3.     for j in range(m)
    l4.         r=r+(i*j)
    l4. print(r)
```

- Total= $l1 + n(l2 + (m(l3 + l4)) + l5$
- Si $n = p$ Total= $an^2 + bn + c \approx an^2$
- Complexité temporelle : $O(n^2)$

Ordre de grandeur

Pourquoi garder $O(n^2)$ comme ordre de grandeur pour

$$T(n) = an^2 + bn + c$$

Exemple: Prenons $n^2 + n + 1$

- si $n = 10$ quelle est la contribution de n^2 dans le résultat final?

$$\frac{n^2}{n^2 + n + 1} = \frac{10^2}{10^2 + 10 + 1} = 0,9 \text{ soit } 90\%$$

- si $n = 100$ quelle est la contribution de n^2 dans le résultat final?

$$\frac{n^2}{n^2 + n + 1} = \frac{100^2}{100^2 + 100 + 1} = 0,99 \text{ soit } 99\%$$

Conclusion : plus n devient grand plus les facteurs autre que n^2 deviennent négligeables.

Boucles While

```
def BoucleWhile(n,m):
  l1. i = 0
  l2. j = 0
  l3. r = 0
  l4. while i < n and j < m:
    l5. r = i + j
```

- Total= maximum entre n et m

En fonction des cas

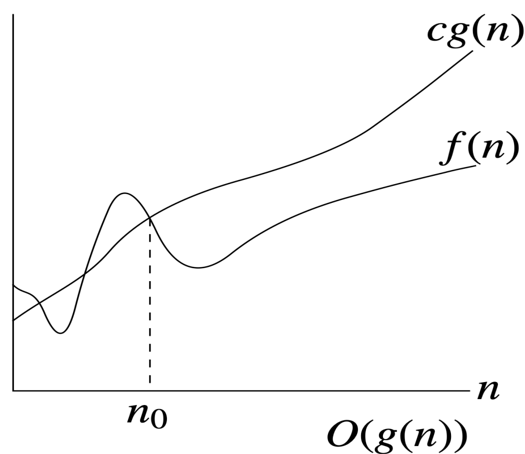
- Soit D_n l'ensemble des configurations possibles pour des données de taille n
- Soit $A(d)$, la complexité de l'algorithme A sur la configuration $d \in D_n$

- Complexité dans ***le meilleur des cas***
  - $\min(d) | d \in D_n$
- Complexité dans ***le pire des cas***
  - $\max A(d) | d \in D_n$
  - Borne supérieure. Permet de choisir une valeur de n maximum.
- Complexité ***en moyenne***
  - $\sum p(d) * A(d) | d \in D_n$
  - $p(d)$, la probabilité d'avoir la configuration d

Notations

- Soit $T(n)$ le temps d'exécution d'un programme en fonction de la taille n caractérisant le nombre d'éléments traités
- On note:
  - O : majorant
  - Ω : minorant
  - Θ : encadrement (ou la moyenne)

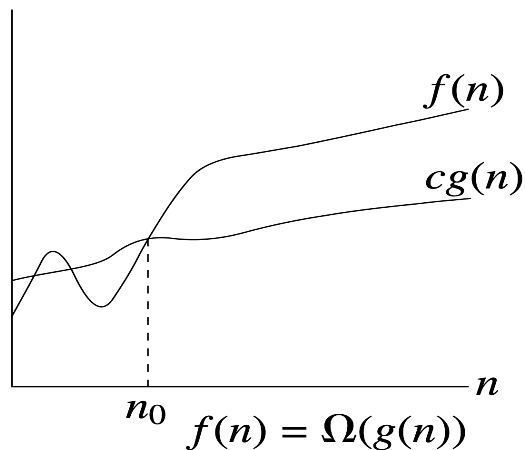
Majorant



Fonction majorant $T(n)$

- Indication sur les performances de l'algorithme dans le pire des cas, pour n très grand.
- $T(n)$ est en $O(g(n))$ s'il existe un entier n_0 et une constante $c > 0$ tels que : $\forall n \geq n_0, T(n) \leq c.g(n)$

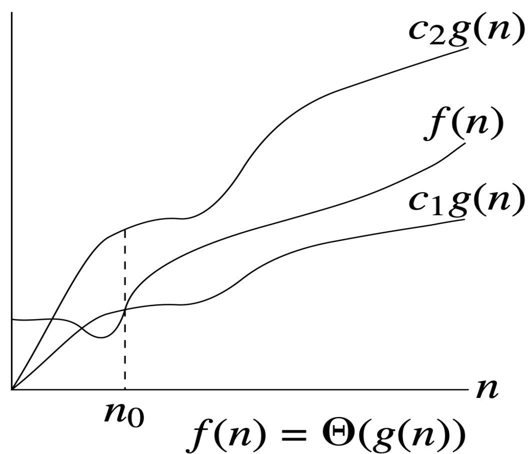
Minorant



Fonction minorant $T(n)$

- Indication sur les performances de l'algorithme dans le meilleur des cas, pour n très grand
- $T(n)$ est en $\Omega(g(n))$ s'il existe un entier n_0 et une constante $c > 0$ tels que : $\forall n \geq n0, T(n) \geq c.g(n)$

Encadrement



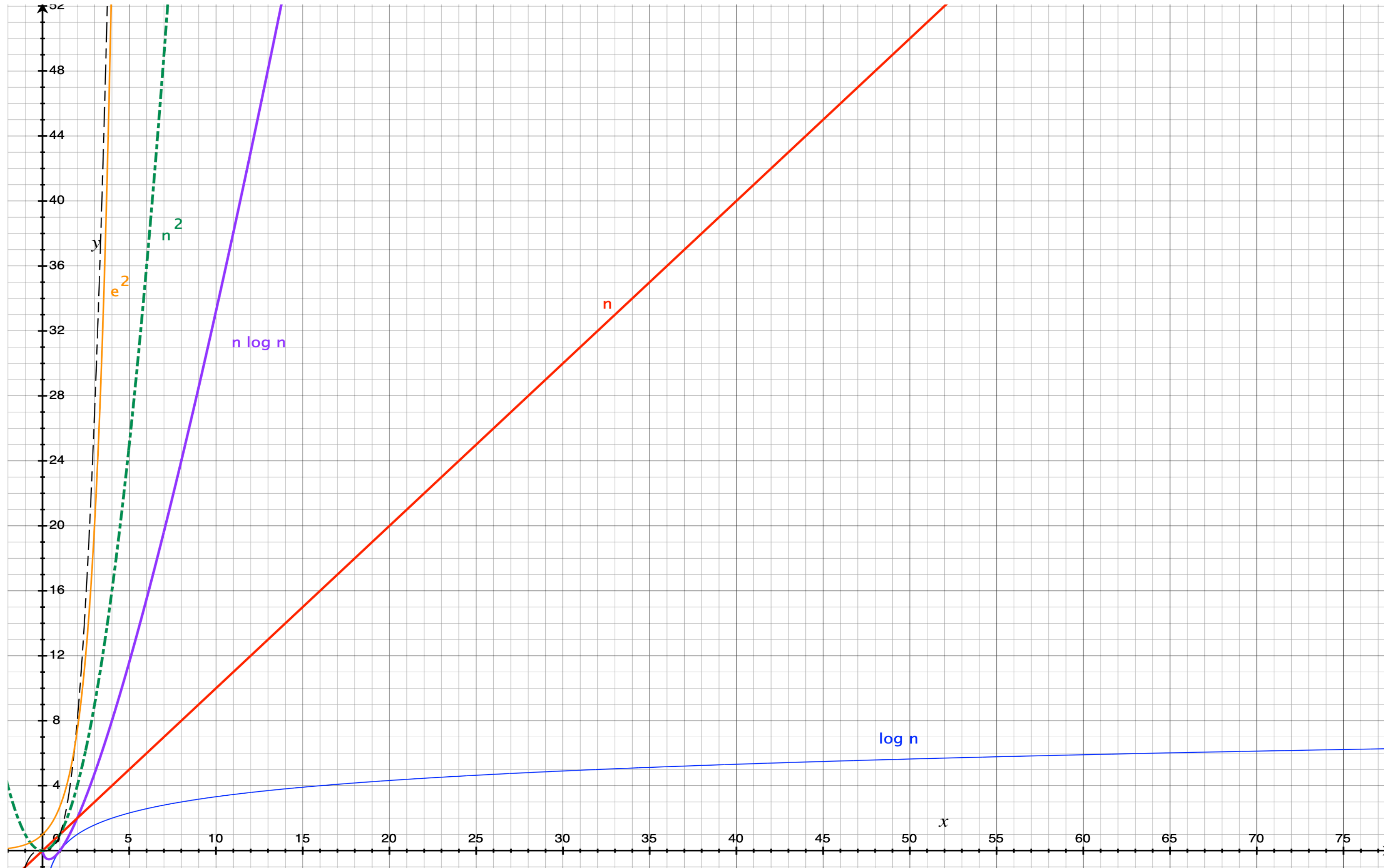
Encadrement de $T(n)$

- Caractérise le fait que l'algorithme a des comportements similaires quelles que soient les données.
- $T(n)$ est en $\Theta(g(n))$ s'il existe un entier n_0 et deux constantes c_1 et c_2 positives tels que $\forall n \geq n0, c1.g(n) \leq T(n) \leq c2.g(n)$

Classes de complexité

Notation	Nom
$O(1)$	Constante
$O(logn)$	Logarithmique
$O(n)$	Linéaire
$O(n.logn)$	
$O(n^2)$	Quadratique
$O(n^3)$	Cubique
$O(2^n)$	Exponentielle
$O(n!)$	Factorielle

Représentation graphique



Nombres d'opérations

n \ O	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	n^4	2^n	$n!$	n^n
1	1	1	1	1	1	1	1	2	1	1
5	1	2	5	12	25	125	625	32	120	3125
10	1	3	10	33	10^2	10^3	10^4	10^3	3.10^6	10^{16}
10^2	1	7	10^2	664	10^4	10^6	10^8	10^{30}		
10^3	1	10	10^3	10^4	10^6	10^9	10^{12}			
10^6	1	20	10^6	2.10^7	10^{12}	10^{18}	10^{24}			

Taille du problème

On suppose que l'opération élémentaire s'exécute en 1 µs

t \ O	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n	$n!$	n^n
1 s	∞	∞	10^6	$6{,}3.10^4$	10^3	100	20	10	7
1 mn	∞	∞	6.10^7	3.10^6	7.10^3	400	26	11	8
1 h	∞	∞	4.10^9	$1{,}3.10^8$	6.10^4	1500	32	12	9
1 jour	∞	∞	9.10^{10}	3.10^9	3.10^5	4400	36	14	11

Nombres d'opérations

On suppose que l'opération élémentaire s'exécute en 1 µs

	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
$n=10^2$	≈1µs	6,6µs	0,1ms	0,6ms	10ms	1s	4.10^{16} ans
$n=10^3$	≈1µs	9,9µs	1ms	9,9ms	1s	17mn	∞
$n=10^4$	≈1µs	13,3µs	10ms	0,1s	100s	11,5j	∞
$n=10^5$	≈1µs	16,6µs	0,1s	1,6s	2,7h	31,7 ans	∞
$n=10^6$	≈1µs	19,9 µs	1s	19,9s	11,5j	317 siècle	∞

Etude de la complexitées: Exemple

Suite de Fibonacci

Algorithme récursif

```
def fiboR(n,t):
    print("fibo(",n,") ->",t)
    if(n==0 or n==1):
        return(1)
    else:
        return(fiboR(n-1,t+"g")+fiboR(n-2,t+"d"))
```

```
fibo( 6 ) -> a
fibo( 5 ) -> ag
fibo( 4 ) -> agg
fibo( 3 ) -> aggg
fibo( 2 ) -> agggg
fibo( 1 ) -> aggggg
fibo( 0 ) -> aggggd
fibo( 1 ) -> agggd
fibo( 2 ) -> aggd
fibo( 1 ) -> aggdg
fibo( 0 ) -> aggdd
fibo( 3 ) -> agd
fibo( 2 ) -> agdg
fibo( 1 ) -> agdgg
fibo( 0 ) -> agdgd
fibo( 1 ) -> agdd
fibo( 4 ) -> ad
fibo( 3 ) -> adg
fibo( 2 ) -> adgg
fibo( 1 ) -> adggg
fibo( 0 ) -> adggd
fibo( 1 ) -> adgd
fibo( 2 ) -> add
fibo( 1 ) -> addg
fibo( 0 ) -> addd
13
```

Suite de Fibonacci itératif

Algorithme itératif avec tableau

```
def fiboIT(n):
    fib = [0 for p in range(0, n+1)]
    fib[0]=1;fib[1]=1
    for i in range(2,n+1):
        fib[i]=fib[i-1]+fib[i-2]
    return(fib)
```

Resultat

```
[1, 1, 2, 3, 5, 8, 13]
```

Algorithme itératif sans tableau

```
def fiboIsT(n):
    a=1;b=1
    r=0
    for i in range(2,n+1):
        r=a+b
        b=a
        a=r
    return(r)
```

Resultat

```
13
```

Suite de Fibonacci

Quel est la meilleure solution?

- La solution récursive?
- La solution itérative avec tableau?
- La solution itérative sans tableau?

Suite de Fibonacci

Quel est la meilleure solution?

- La solution récursive?
  - Complexité temporelle: $O(n^2)$
- La solution itérative avec tableau?
  - Complexité temporelle: $O(n)$
- La solution itérative sans tableau?
  - Complexité temporelle: $O(n)$

Suite de Fibonacci

Quel est la meilleure solution?

- La solution récursive?
  - Complexité temporelle: $O(n^2)$
  - Complexité spatiale: $O(1)$
- La solution itérative avec tableau?
  - Complexité temporelle: $O(n)$
  - Complexité spatiale: $O(n)$
- La solution itérative sans tableau?
  - Complexité temporelle: $O(n)$
  - Complexité spatiale: $O(1)$

Livre conseillé