

Introduction à l'Algorithmique

TC1-1A-ECL

Ecole Centrale de Lyon
Département Mathématiques-Informatique

Cours TC1-Chapitre 1

Introduction à l'Algorithmique et aux structures de données

(CHAPITRE I)

Algorithmes

2017-2018

Plan général des actions TC

- TC1 : introduction à l’algorithmique avec Python
- TC2 : Programmation objets avec Python,
Programmation événementielle (GUI)
- TC3 : Projet WEB

Rappel Plan Général du cours

① Chapitre 1

- ▶ Quelques algorithmes remarquables
- ▶ D'un problème à sa solution
- ▶ Les machines et leurs langages
- ▶ Découverte de Python : quelques exemples
- ▶ Notions de Complexité
- ▶ Récursivité & Induction Mathématique, Transformation

② Chapitre 2

- ▶ SD remarquables : Liste / Tableaux, Graphes, Arbres
- ▶ Outils de spécification : TDA
- ▶ Programmation Dynamique (PrD)
- ▶ Algorithmes à essais successifs (AES)
- ▶ Autres schémas algorithmiques importants

Google page Ranking

- Google détient 67% du marché de recherche d'information sur internet (18% pour microsoft *Bing*, *Yahoo* 11%, le reste pour *Ask*, *AOL*, etc.).
- L'algorithme de Google travaille avec des *spiders* (ou *crawler* : araignée / robot d'indexation) et maintient un très grand indexe de mots clés + url.
- Critère principal : le nombre et la qualité des liens vers les pages → l'importance du site
- Le raisonnement : plus un site est référencé, plus il est susceptible d'en recevoir !
- Autres critères : le choix de la page par les internautes dans les résultats, la fréquence et la position des mots clés dans 1 page, la durée de vie de la page, etc.
- La pondération de chacun de ces critères connus (et inconnus ?) fait la force de Google (biaisé néanmoins par des considérations commerciales).

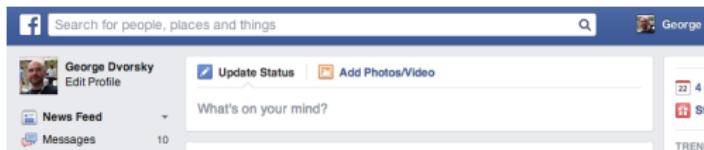
Google page Ranking (suite)

Les sources de Google (*Sergueï Brin, Larry Page*) :

- Au départ, une idée sur les indices de citations (**sociologie** : G. Pinsky et F. Narin - 1976) a inspiré Google.
 - Ils ont appliqué cette idée aux pages web (les liens In / Out des pages)
- Cette idée est formalisée en **Mathématiques / Algèbre** par les (très grandes) matrices + calculs de valeurs propres exploitée par le théorème de Perron-Frobenius (1900, application en chaînes de Markov et en théorie de Graphes).
 - Calcul de *pagerank* (Google)
- Un système **Informatique** distribué efficace et adapté implante et prend cet ensemble en charge .

Facebook's News Feed

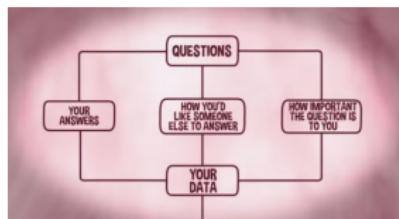
- Pré-sélectionne (pour nous) les articles choisis par (l'algorithme de) Facebook (sauf si vos préférences demandent de tout afficher dans l'ordre chronologique).



- Pour sélectionner ces articles, l'algorithme de Facebook s'appuie (entre autres)
 - sur le nombre de commentaires,
 - qui a posté l'article suivant une classification des personnes plus ou moins populaires (ou ceux avec qui vous avez eu des contacts ou ceux dont vous avez déjà lu des choses),
 - le type de l'article (photo, vidéo, état, MAJ, etc.), ...
- Pour placer des pubs contextuels, Google / Facebook / ... utilisent également des algorithmes pour épier nos habitudes, usages, achats, questions posées et les mots choisis.

OKCupid Date Matching

- Un marché de 2 milliards avec une progression constante de 3% depuis 2008 !
- Les gens n'ont plus le temps de trouver l'âme soeur comme avant et 1/3 des mariages serait du à ces sites.
- On prétend que ces couples sont plus solides que les 'classiques'.
- La méthode utilisée par *OKCupid* est basé sur le *matching* des préférences, centres d'intérêts, tendances, gouts, ...



- Mais au lieu de faire du matching brute des intérêts communs, OKCupid analyse les réponses de chacun, attache à ces réponses une pondération suivant son importance pour la personne ainsi que pour un/une âme soeur.

NSA Data Collection, Interpretation, and Encryption

- Vous croyez être surveillé par des caméras ? Non, vous êtes épiés par des algos.
- *Edward Snowden* a dévoilé que la NSA surveille les données de milliards de personnes.
- Ses révélations ont montré qu'il existe des algorithmes d'analyse des données de surveillance récupérées par 5 pays : USA, CA, GB, Aust., Nlle-Z.
- Données provenant des écoutes tél-ques, mails, webcams et vidéo, Géo loc, ...
- La NSA est drôle : elle dit ne pas collecter d'info puisque par (leur) définition :

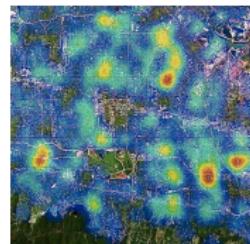
 une information n'est collectée que si elle est effectivement analysée et transformée sous une forme intelligible par un membre de la NSA pendant ces missions officielles.

→ C-à-d. quelqu'un qui possède chez lui une bibliothèque pleine de livres ne collecte pas de livres. Les seuls livres qu'il aurait collectés sont ceux qu'il a déjà lus !

- Le problème : collecter des infos nous concernant et à notre insu, quand bien même toutes ces données ne sont pas encore exploitées.

IBM'CRUSH

- Pas encore trop rependu mais de plus en plus de départements de police aux USA utilisent un outil d'*analyse prédictive* (cf. le film *Minority Report*).
- Le CRUSH d'IBM (*Criminal Reduction Utilizing Statistical History*) aurait permis à la police de Memphis de réduire les crimes de 30% depuis 2006 (chiffres publiés).
→ (D'autres états US et plusieurs autres pays sont fortement intéressés.)



- CRUSH exploite les techniques statistiques et des algorithmes de déduction et de prédiction (voir <http://www.ibm.com/smarterplanet/us/en/leadership/memphispd/>)./..

IBM'CRUSH (suite)

- En matière de la lutte prédictive contre le crime (Crime Predictive Fighting), CRUSH permet d'analyser les *incidents* pour prédire les *points chauds* où un crime est susceptible d'avoir lieu
 - Pour déployer les ressources et les hommes adéquats à ces endroits.
- Ces algorithmes seront (bientôt) déployés pour surveiller les surfs sur Internet, les données GPS, celles des PDAs et smartphones (écoute téléphonique), la signature biologique (e.g. de la rétine) pour une analyse en temps réel.

Autres algorithmes remarquables

- Algorithmes de **Tri** (cf. complexité),
 - Traitement et utilisation du signal (**FT**, **FFT**),
 - **Cryptographie** (e.g. RSA), Signatures sécurisées (Hash MD5), **Factorisation** de grands entiers (cf. cryptage sécurisé),
 - Détection de **communautés** (Graph Link Analysis),
 - **Compression** (audio : MP3, AAC, OGG, PCM, Vidéo : DV, Mpeg DivX, Xvid, ...),
 - Génération de nombres **aléatoires**,
 - Algorithmes de **graphes** (Dijkstra, Flow, MST),
 - Calculs matriciels avancés,
 - etc.
- Liens avec le **Big data**, les contraintes (3V) ?

Les Robots exécutent des algorithmes

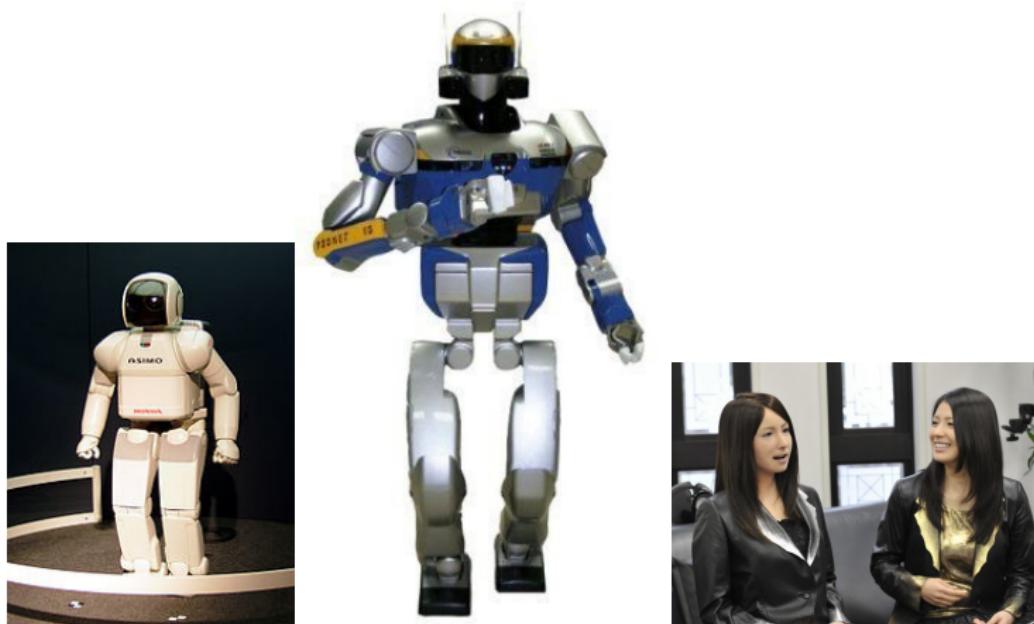
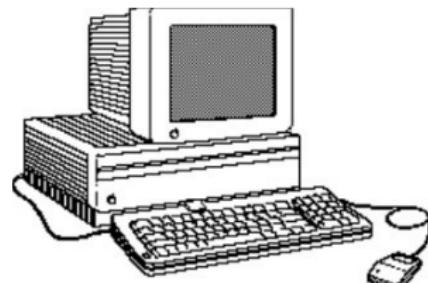


FIGURE 1 : Robots Azimov, HRP & Ucroa [thanks to Futura HighTech]

Machines et Algorithmes

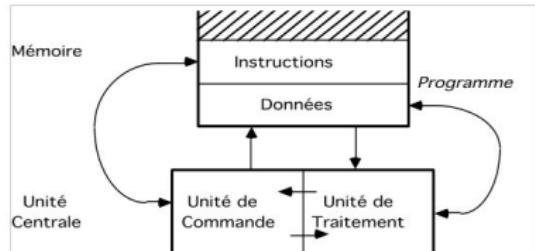
- Un ordinateur :

- Un objet de décoration ?
- Un moyen de calcul ?
→ Mémoire et l'Unité Centrale
- Architecture *Von Neumann*



- Comment cela fonctionne ?

- Matériel / Logiciel, OS
- Est-ce intelligent ?
- Que peut il faire ?
- Algorithmes (Vitesse, Volume, Complexité, ...)



Ordinateur (modèle *Von Neumann*)

Machines et Algorithmes (suite)

- Nous et les ordinateurs
 - Un outil de travail : faire tourner des applications
 - programmer un calculateur
 - Un moyen de traitement d'**information**.
- **Information** : données et connaissances
- Traitement de l'information : **réduction** de la quantité d'information.
 - à l'aide d'un algorithme (un programme) P , réduire l'espace de recherche en un espace des résultats puis à celui qui nous intéresse

Données *Programme P* Résultats

- Programmer un ordinateur : communiquer avec dans un langage qu'il comprend
- ☞ Voir en Annexes un exemple de *réduction* (section 1 page 43 du cours 1)

Machines et Algorithmes (suite)

- Exemple d'algorithme (trivial) : *madame Dupont veut faire des frites*
 - chercher le panier à la cave, sortir la poêle du placard
 - éplucher les pommes de terre, ranger le panier, allumer le feu, ...
 - Autres schémas simples d'algorithmes :
 - Patrons de tricot, Recettes de cuisine,
 - Indication d'un chemin (1er à gauche, 2nd à droite, tourner..)
 - A un problème, on trouve une solution Algorithmique, Équationnelle, Logique ou Ad-hoc (bio-inspiré cf. génétique, neurones, ...heuristique) : **besoin d'algorithmes dans tous les cas !**
- N.B. : la démarche Algorithmique est différente de la démarche Équationnelle (ou celle Mathématique, par la construction d'un système d'équations / inéquations).
- En Programmation Linéaire (ou Logique), le *contrôle* (algo) est déjà dans le solveur !
 - ☞ Voir aussi en Annexes (sec. 4 page 47) le cycle de vie des algorithmes.

Machines et Algorithmes (suite)

Quand on écrit un algorithme,

- On décrit une séquence de phrases portant sur des objets de calcul (valeurs).
- Par des expressions choisies dans un répertoire fini et bien défini d'actions élémentaires, identifiées et réputées réalisables a priori.
- Expressions indépendantes des langages de programmation.

Ex. : de $S = \{x \mid \text{premier}(x), 0 < x \leq k \in \mathbb{N}\}$ aux instructions de base.

→ Les instructions (le code) qui construisent S ne sont pas toujours uniques.

- Parfois, une abstraction peut être re-développée par un algo : cf. pour créer S .
 - On décrit une suite générale et totalemen definie d'opérations et de leurs enchaînements ayant pour but le traitement (et la transformation) d'informations.
- ☞ On peut mettre en place une (méta) stratégie (et/ou heuristiques), P. Ex. : Gloutonne, AES, B&B, A*, PrD, Diviser / Séparer-pour-Régner, ...

L'importance des Structure des données

- Un **algorithme** = un raisonnement logique (*réduction*) contrôlée

Algorithme = Logique + Contrôle

une logique + stratégie de mise en oeuvre

Programme = Algorithme + Structure de données (SD)

- Pour arriver à un programme (dans un langage de programmation *L*) :
 - On choisit les Structures de Données (SD = produit cartésien de domaines)
 - On traduit l'algorithme (et la SD) en une séquence d'actions dans le langage *L* accepté par une machine sur laquelle il doit être exécuté.
 - Le répertoire de ces actions est fini et bien compris (**sémantique**).
- Ces schémas seront traduits à leur tour en actions plus simples (pour la *machine*).
- On s'intéresse aux processus séquentiels = bloc séquentiel de commandes
- L'effet principal d'un programme se résume à modifier l'état de la mémoire

Classification des algorithmes

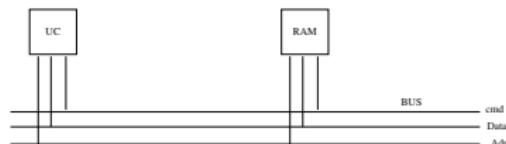
Variation de la Logique et/ou du Contrôle

- Par **implantation** :
 - ▶ Récursif , Itératif
 - ▶ Logique , Procédurale (couvre "Fonctionnelle" au sens non-logique)
 - En **Programmation Logique**, le *contrôle* (la déduction) est prédéfinie, on exprime seulement les *axiomes*
 - ▶ Séquentiel, Parallèle (et Distribué)
 - ▶ Déterministe , Non-déterministe
- Par **stratégie** (paradigme) de conception :
 - ▶ Diviser-pour-régner (e.g. Merge-Sort), Séparer-pour-régner (e.g. *Covering*)
 - ▶ Programmation Dynamique (PrD : Divide/Seperate & Conquer avec stockage = recurse & reuse)
 - ▶ Best First, Best Fit, First Fail, A*,
 - ▶ Programmation Linéaire (ou Programmation Mathématique)
 - ▶ Probabiliste (Stochastique) / Heuristique / Bio-inspirée / ...
- Par **complexité** : de *constante* à $N!$ (voire, plus)

Anatomie d'une instruction

- Exemple intuitif de passage du code d'une affectation au code *Assembleur* :
 - Ce qui se passe lors de l'exécution de $N \leftarrow N + 1$ sur une machine ?
- L'instruction est décomposée en code Assembleur (sur un processeur simple) :

| | |
|----------------|----------------------------|
| <i>Load N</i> | [Accu \leftarrow N] |
| <i>Add #1</i> | [Accu \leftarrow Accu+1] |
| <i>Store N</i> | [N \leftarrow Accu] |



- Ce code doit être préparé pour être exécuté :

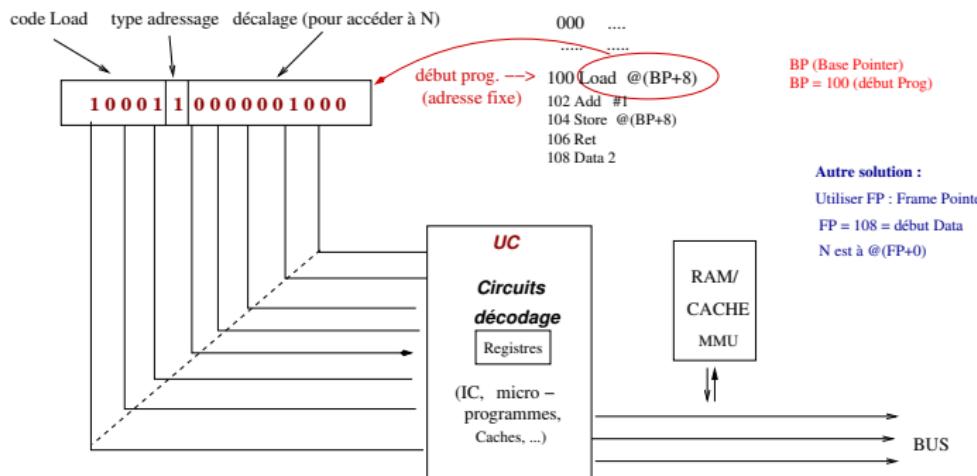
| | | |
|------|-----------------|---------------------------------------|
| 0000 | <i>Load +8</i> | <i>0000 marque 'le début' relatif</i> |
| 0002 | <i>Add #1</i> | |
| 0004 | <i>Store +4</i> | |
| 0006 | <i>Ret</i> | <i>Retour à l'appelant</i> |
| 0008 | <i>Data 2</i> | <i>place de N</i> |

- ☞ Une idée : éviter des '+4' et '+8' ? Référencer toujours N par (*début du code* + 8).
 - On place *début de code* dans un registre appelé BP

Différents niveaux de traduction

Décodage d'une instruction (*Load*) chargée dans le registre d'inst. de l'UC :

- Ce programme est chargé à l'adresse mémoire 100 (= début du code, placé dans *BP*)



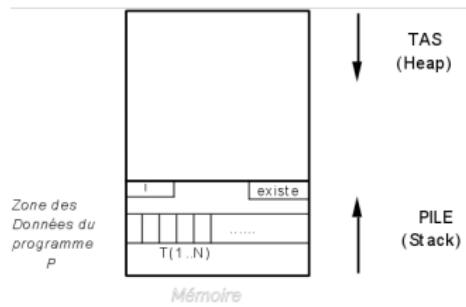
- ☞ Voir un autre exemple en Annexes du cours 1 (sec. 2, page 45)

Çà tombe "Pile" ?

- Contexte et Pile d'un programme P

(simple) :

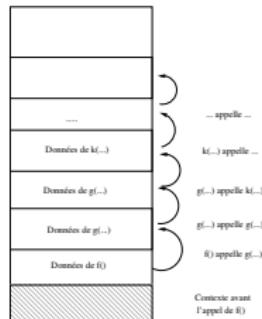
- un tableau $T[1..N]$
- une booléenne existe
- un entier i
- des appels de fonctions



- Quand on appelle une fonction qui appelle une fonction (peut-être elle-même) qui appelle ...

Et quand la pile est pleine :

→ erreur "stack overflow"



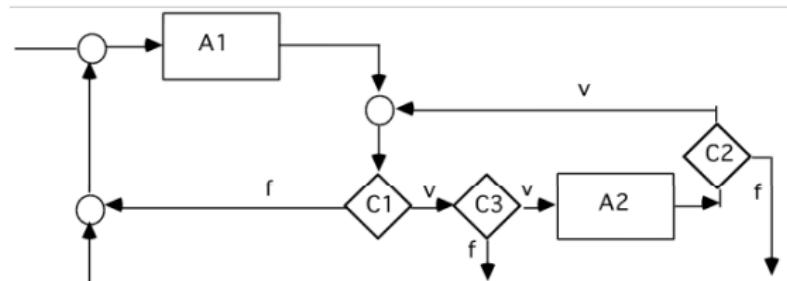
Formalismes et Langages de programmation

- Langage de description d'algorithmes : graphique / textuel

Graphique : organigramme

- Formalisme algorithmique primitif permettant la description des enchaînements
- Expression d'algorithme à l'aide de symboles graphiques

Exemple :

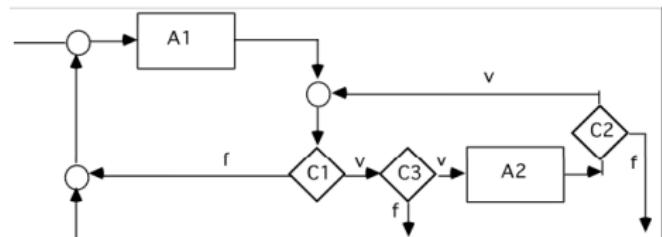


- Il existe d'autres formalismes graphiques (RdP, Grafcet, Building Blocks).

Formalismes et Langages de programmation (suite)

Formalismes Textuels

```
étiquette1 : A1
étiquette2 : si C1 alors
    si C3 alors
        A2;
        si C2 alors goto étiquette2
        sinon goto fin
        fin si
    sinon goto étiquette1
    fin si;
sinon goto étiquette1
fin si
fin :
```



Formalismes et Langages de programmation (suite)

Autres exemples

- Le langage Fortran II ne connaissait pas le schéma *si-alors-sinon*.
 - Si $a=5$ Alors $a := a+1$ Sinon Si $a < 5$ alors I2 Fin si ; Fin si ; I3 ;

```

début : si a>5 alors goto I3
        si a<5 goto I2
        a:=a+1
        goto I3
I2 :      .....
I3 :      .....
  
```

- Traduction de $\mathbf{z} = \mathbf{x} * \mathbf{y}$ en termes d'addition (x et y positifs)

```

début : z=0
mult : si y < 0 alors goto fin
        z := z + x
        y := y - 1
        goto mult
fin :   ...
  
```

Quelques schémas algorithmiques (importants)

- Schéma itératif **Tant que** :

Tantque Cond faire

A

Fin faire

*Tant qu'il y a quelque chose sur la bande faire
établir la fiche d'impôts
passer au contribuable suivant
Fin Tant que*

- Schéma itératif **Répéter ... Jusqu'à** :

Répéter

A

Jusqu'à Cond

*Répéter
Prendre température
Jusqu'à température > 50
Déclencher alarme*

Quelques schémas algorithmiques (importants) (suite)

- Schéma itératif **Pour** :

Pour indice dans itérable

A

Fin Pour

```
Pour i dans 1 .. 10 faire
    écrire t[i]
fin Pour
```

- Schéma itératif **combiné** :

Pour indice dans itérable Jusqu'à Cond faire

A

Fin Pour

- Exemple : quête de 100 francs (non équitable !)

```
S <- 0;
Pour X dans la liste_employés Jusqu'à S >= 100 faire
    S <- S + don(X)
Fin pour
```

Propriétés des algorithmes

- Algorithme effectif et traçable (*tractable* = calculable) :

Il doit pouvoir être effectivement réalisé par une machine (si l'on peut faire de même avec un papier et un crayon en un temps fini).

- Quelle que soit la donnée sur laquelle il travaille, un algorithme doit toujours se terminer après un nombre fini d'opérations, et fournir un résultat (partiel, message, ...) prévu.

- On considère les algorithmes déterministes : toute exécution d'un tel algorithme sur les mêmes données donne lieu à la même suite d'opérations.

- Un algorithme doit être juste et complet (+ question d'*unicité*, ...)

→ Tout ce que l'algorithme calcule est juste (*justesse*)

→ Il sait calculer (toutes) les réponses justes (*complétude*)

☞ Preuve ?

*Algorithme = Logique + Contrôle** (le sens + la stratégie)

Programme = Algorithmes + structures de données

* Pour une même logique de l'algorithme, l'efficacité (stratégie de contrôle) peut être modifiée

Décomposition des structures de données

- La manipulation des (structures de) données et leurs décompositions :

- Décomposition **fonctionnelle** (vue et définition externe) :

Retrait : Compte × entier → Compte

- Décomposition **logique** (plus de détails) :

l- titulaire : *(nom × prénom × adr)*

- Description de la donnée l- valeur : *entier*

Compte l- date création : *(jour × mois × an)*

- Description de l'action (algorithme) : **Retrait**

- Décomposition **physique** :

- Décision d'implantation physique en termes de tableaux, listes...

☞ Nos algorithmes traduisent ces décompositions (en particulier Physique)

→ Mise en oeuvre Algo-SD : *séparée* ou *encapsulation* (Voir TC2 programmation objets)

Décomposition des structures de données (suite)

Un exemple : Horner

- Évaluation du polynôme (la base X et les a_i connus) :

$$P_n = a_n X^n + \dots + a_2 X^2 + a_1 X + a_0$$

- Une solution (stratégie de contrôle) : réécrire le polynôme

$$P_n = (\dots((a_n * X + a_{n-1}) * X) * X \dots + (a_i) * X) \dots * X) + a_0$$

Pour $n = 4$: $P_4 = (((((a_4) * X + a_3) * X + a_2) * X + a_1) * X + a_0)$

- Exemple : que représente en base 10, le nombre 5732 en base 8

→ On a $n = 3, X = 8$

$$5 = 5 \text{ en base 10}$$

$$(5 * 8) + 7 = 47 \text{ en base 10}$$

$$(47 * 8) + 3 = 379 \text{ en base 10}$$

$$(379 * 8) + 2 = 3034 \text{ en base 10}$$

Décomposition des structures de données (suite)

- Symétriquement, si les indices vont de 0 à n (de gche. à dte.) :

$$P_n = a_0 X^n + \dots + a_{n-2} X^2 + a_{n-1} X + a_n \quad (a_k X^{n-k} : \text{somme} = n)$$

$$P_n = (\dots((a_0 * X + a_1) * X) * \dots + (a_{n-i}) * X) \dots * X + a_n$$

- Description fonctionnelle :** $P_0 = a_0$

$$P_n = P_{n-1} * X + a_n$$

- Description logique :** X connu, a_i donnés, calculer P_n

```

P := a0;
i := 1;
Tant que i <= n faire      (les indices vont de 0 à n)
    P := P * X + ai;
    i := i + 1;
Fin Tant que;

```

- Description physique :**

Exemple du programme Python (schéma **for**) où une liste contient le vecteur a

```

# Le vecteur a et la valeur de X sont donnés
P = a[0];
for i in range(1,n+1) :
    P = P * X + a[i];

```

Éléments du langage Python

- Pourquoi Python ?
- Langage impératif, fonctionnel et objet
 - Impératif : dispose de l'effet mémoire (affectation)
 - Fonctionnel comme Lisp ou CAML mais sans le typage (ni ref !)
 - L'absence du typage peut poser problèmes
 - Objets
- Types en Python
 - types de base
 - liste, ensemble, tuple, dict
- Expressions en Python
- Exceptions
- Fonctions
- Bibliothèques importantes

Les Tours de Hanoï

Historique : un jour, dans le grand temple de Bénarès (Royaume du Siam), au dessous du dôme qui marque le centre du monde, Dieu planta trois aiguilles de diamant sur une dalle d'airain. Sur une de ces aiguilles, Il enfila au commencement des siècles, 64 disques d'or pur, le plus large reposant sur l'airain, et les autres, de plus en plus étroits, superposés jusqu'au sommet. C'est la **tour sacrée du Brahmâ**.

Depuis, nuit et jour, les prêtres se succèdent sur les marches de l'autel, occupés à transporter la tour de la 1e aiguille sur la 3e, sans s'écartez des règles fixes qui ont été imposées par Brahma : *Ne jamais mettre un plus grand disque sur un petit ; Ne déplacer qu'un disque à la fois.*

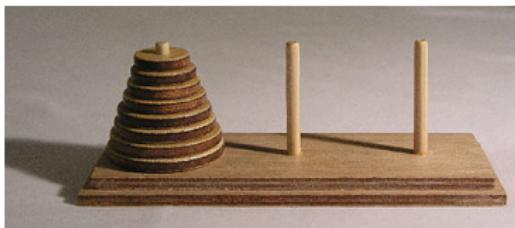
Quand tout sera fini, les tours (et les brahmanes) tomberont, et ce sera la fin des mondes !

Quand ? : un jeu à 64 disques requiert un minimum de $2^{64} - 1$ déplacements.

Si on déplace un disque par seconde → 86 400 déplacements par jour → environ 213 000 milliards de jours

→ à peu près **584,5 milliards d'années**, → soit 43 fois l'âge estimé de l'univers (env. 13,7 milliards d'années).

Les Tours de Hanoï (suite)



```

def hanoi(n, dep, arr, aux) :
    if n > 0 :
        hanoi(n-1, dep, aux, arr)
        print(dep, '--->', arr, end=' ', )
        hanoi(n-1, aux, arr, dep)

n=5
hanoi(n, 'A', 'B', 'C')

```

- Trace pour $n = 5$

$$\begin{aligned}
 A &\longrightarrow B, A \longrightarrow C, B \longrightarrow C, A \longrightarrow B, C \longrightarrow A, C \longrightarrow B, A \longrightarrow C, \\
 B &\longrightarrow C, B \longrightarrow A, C \longrightarrow A, B \longrightarrow C, A \longrightarrow B, A \longrightarrow C, B \longrightarrow C, A \longrightarrow B, \\
 C &\longrightarrow A, C \longrightarrow B, A \longrightarrow B, C \longrightarrow A, B \longrightarrow C, B \longrightarrow A, C \longrightarrow A, C \longrightarrow B, \\
 A &\longrightarrow B, A \longrightarrow C, B \longrightarrow C, A \longrightarrow B, C \longrightarrow A, B \longrightarrow C, A \longrightarrow B,
 \end{aligned}$$

Palindrome

- Mots / phrases qui se lisent dans les deux sens.

```
def palindrome(mot) :
    l = len(mot)
    i=0
    contin = True
    while i < l//2 and contin :
        contin = (mot[i]==mot[l-i-1])
        i += 1
    return(contin);

# print(palindrome('laval'))
# True

print(palindrome('toto'))
# False
```

- La version récursive :

```
def palindrome_rec(mot) :
    if len(mot) < 2 : return True
    return mot[0] == mot[len(mot)-1] and palindrome_rec(mot[1:len(mot)-1])
    # mot[1:len(mot)-1] car (len(mot)-1) ne sera pas dans le range.

# print(palindrome_rec('laval'))
# True

# print(palindrome('toto'))
# False
```

Palindrome (suite)

- Et pour une phrase contenant des espaces : on enlève d'abord les espaces :

```
ph= 'engage le jeu que je le gagne'
ph1=mot.replace(' ', '') # On enlève les espaces
print(palindrome_rec(ph1))
# True
```

- Un autre exemple : *Henri Tournier* a rapporté des photographies d'un palindrome dans l'Église de Corps en Isère.

On y lit sur les 4 côtés du bénitier :

"nipson anomêmata mê monan opsin"

→ (lave mes péchés et non mes seuls yeux).



```
ph= 'nipson anomêmata mê monan opsin'
ph1=mot.replace(' ', '') # On enlève les espaces
print(palindrome_rec(ph1))
# True
```

Le mot le plus long

- Rechercher le mot le plus long dans un texte.

```
def mot_le_plus_long(lst) :  
    lg=0  
    mot_long=''  
    for mot in lst :  
        if len(mot) > lg :  
            lg=len(mot)  
            mot_long=mot  
    return mot_long  
  
ph='elu par cette crapule'           # Encore un palindrome !  
lst_mots=ph.split()                  # Liste de mots de la phrase  
  
print(mot_le_plus_long(lst_mots))  
# crapule
```

Chiffres Romains

- Trouver le nombre décimal qui correspond à un nombre romain.
- La logique : la valeur de toute lettre romaine s'additionne si la lettre suivante correspond à une valeur plus petite (cf. $XI=11$) sinon, elle se soustrait (cf. $IX=9$)
- Une représentation physique qui suite directement la logique

```
_1984 = 'MCMXXXIV'  
_2015 = 'MMXV'  
_1968 = 'MCMXLVIII'  
  
chiffre_romain = ['I', 'V', 'X', 'L', 'C', 'D', 'M']  
romain_to_entier = [1, 5, 10, 50, 100, 500, 1000]  
  
# R = _1984  
# R=_1968  
R = _2015  
  
val_entiere=0  
for I in range(len(R)) :  
    if I < len(R)-1 and  
        romain_to_entier[chiffre_romain.index(R[I])] < romain_to_entier[chiffre_romain.index(R[I+1])] :  
            val_entiere-= romain_to_entier[chiffre_romain.index(R[I])]  
    else : val_entiere+= romain_to_entier[chiffre_romain.index(R[I])]  
print(val_entiere)
```

Chiffres Romains (suite)

Variante : on conserve la même logique, mais on modifie la représentation physique :

- On utilise un *while*, accès à la lettre romaine suivante d'une manière différente, on utilise les exceptions de Python pour contrôler l'exécution.

```
_1984 = 'MCMXXXIV'  
_2015 = 'MMXV'  
_1968 = 'MCMXLVIII'  
chiffre_romain = ['I', 'V', 'X', 'L', 'C', 'D', 'M']  
romain_to_entier = [1, 5, 10, 50, 100, 500, 1000]  
  
R = _1984      # R=_1968 # R = _2015  
  
val_entiere=0  
RR=iter(R); I=next(RR)          # I pointe le première élément  
while True :  
    try :  
        N = next(RR)            # Fait avancer l'élément pointé  
        if romain_to_entier[chiffre_romain.index(I)] < romain_to_entier[chiffre_romain.index(N)] :  
            val_entiere-= romain_to_entier[chiffre_romain.index(I)]  
        else : val_entiere+= romain_to_entier[chiffre_romain.index(I)]  
    I=N  
    except StopIteration :      # Accès au suivant impossible : on vient de traiter le dernier  
        val_entiere+= romain_to_entier[chiffre_romain.index(I)]  
        break  
  
print(val_entiere)
```

☞ Un mot sur les *Exceptions* !

Aspects fonctionnels : Fréquences des lettres

- *map, filter, reduce, lambda-calculs, listes / ensemble en compréhension, ...*
- Algorithme de calcul des fréquences des lettres (et de l'espace, pour éviter de les supprimer !) dans un texte à l'aide d'un Dictionnaire Python.

```

def incrementer_nbr_de_lettre(l) :
    """ on fait +1 sur le nombre d'occurrences de la lettre l """
    global d
    d[l] += 1

# Création du dico initial (0 pour tout le monde)
d={l:0 for l in 'abcdefghijklmnopqrstuvwxyz '}

# Transformer la phrase en minuscule et appliquer incrementer_nbr_de_lettre à chaque lettre/espace
# la variable globale 'd' (mutable) est modifiée sur place.
list(map(lambda l : incrementer_nbr_de_lettre(l), "In girum imus nocte et consumimur igni ".lower()))

# Afficher les résultats
print(dict((key,value/100) for (key,value) in d.items()))
# {'x': 0.0, 'm': 0.04, 'h': 0.0, 'o': 0.02, 's': 0.02, 'w': 0.0, 'k': 0.0, 'j': 0.0, 'b': 0.0,
# 'e': 0.02, 'd': 0.0, 'c': 0.02, 'u': 0.04, 'g': 0.02, 'v': 0.0, 'l': 0.0, 'r': 0.02, 'z': 0.0,
# 'i': 0.06, 'f': 0.0, ' ': 0.07, 'n': 0.04, 'a': 0.0, 'y': 0.0, 't': 0.02, 'q': 0.0, 'p': 0.0}

```

a. Encore un palindrome : "nous tournons en rond dans la nuit et sommes consumés par le feu"

Type et structures de données

- Types élémentaires (de base) : bit, octet, char, entier, réel, string ...
- Tuples : n-uplet (couple, triplet, quadruplet, ...), nbr. complexe
- Listes : formes multiples, Tableaux, Pile, File, ...
- Ensembles
- Table de Hachage (Map, Dictionnaires)
- Séquences et Fichiers
- Graphes : orienté ou non, valué ou non, (fortement) connexe, bi-parties,
- Arbres : n-aire, binaire , ABOH, AVL, ...
- B-arbres, Forêts, Treillis,
- ...
- La chaîne de caractères (*string*) est représentée sous forme d'un tableau/liste.

Type et structures de données (suite)

- Il existe d'autres types réalisables à l'aide des types ci-dessus (p.ex. Ball-Tree)
- La plupart de ces types se décline(nt) en sous-types et variants, en particulier les graphes et les arbres.
- **Le choix d'un type de données** (penser à la maintenance/évolution du code) :
 - Se fait suivant les ressources, les besoins (de l'algo), la complexité (espace/temps),
 - Par Exemple un *Tableau* comme SD contigüe, une fois rempli séquentiellement, est destiné plutôt à la consultation plutôt qu'aux insertions fréquentes (il faudra repousser les éléments).
 - Dans une liste : $\text{suivant}(\text{info}_i) = \text{info}_{\text{suivant}(i)}$, pour un tableau : $\text{suivant}(\text{info}_i) = \text{info}_{i+1}$
 - ➔ Ou on prendra un arbre binaire (ABOH) plutôt pour la recherche ($O(\log(N))$).
 - ➔ Même si l'insertion y est possible, le cout peut en devenir important (Niveau = $\log(N)$).
- Important : la représentation Φ ne devrait pas intervenir dans le choix de la SD.
 - ➔ P. ex. choisir entre Liste et Tableau même si on sait que leur représentation ϕ est identique.
- Outil de spécification algébrique de type de données : TDA.

Addendum : Un exemple de réduction

- Un exemple de recherche de Concepts par Généralisation en apprentissage :
 - La théorie est belle mais non praticable !
 - Il nous faut un algorithme qui réduit cet espace et nous mène à une solution.

Énumération de l'espace de concepts (de Versions)

Une définition théorique qui permet de comprendre.

- Pour apprendre (des concepts), on maintient deux ensembles *consistants* :
 - L** (least) : les descriptions *les plus spécifiques* qui couvrent tous les exemples positifs et aucun exemple négatif (spécifique).
 - G** (greatest) : les descriptions *les plus générales* qui ne couvrent aucun exemple négatif mais tous les exemples positifs (générale).
- **On a juste besoin de maintenir à jour L et G**

Addendum : Un exemple de réduction (suite)

Exemple : soit le vocabulaire couleurs $\in \{\text{rouge}, \text{vert}\}$, animaux $\in \{\text{vache}, \text{poule}\}$

| Ex. positifs | Ex. négatifs | L | G |
|----------------|----------------|------------------|--------------------------|
| <> | <> | { } | {<*, *>} (a priori) |
| <vache, verte> | | {<vache, verte>} | {<*, *>} |
| | <poule, rouge> | {<vache, verte>} | {<*, verte>, <vache, *>} |
| <poule, verte> | | {<*, verte>} | {<*, verte>, <vache, *>} |

- Si l'on ajoute <vache, rouge> :
 - Si ex. positif : ajouter <vache, *> à L
 - Si ex. négatif : retirer <vache, *> de G
- Le modèle appris sera *inconsistance* si les exemples positifs et négatifs se contredisent.
- L'ensemble G dénote les phrases qu'on pourrait construire sur cet alphabet.
- Définition théorique (*génératrice*) claire mais incalculable (si passage à l'échelle)
- **Juste mais très coûteux** (penser au Tri et Permutation)
 - Est rendue praticable par des algorithmes (+ heuristiques)

Addendum : Anatomie d'une instruction

Un autre exemple (et sa traduction Assembleur) : soit le programme

```
entier M, N ← 0;
```

```
M ← N+1;
```

```
afficher M;
```

- Machine à registres ; instructions / adresses / données sur 2 octets :

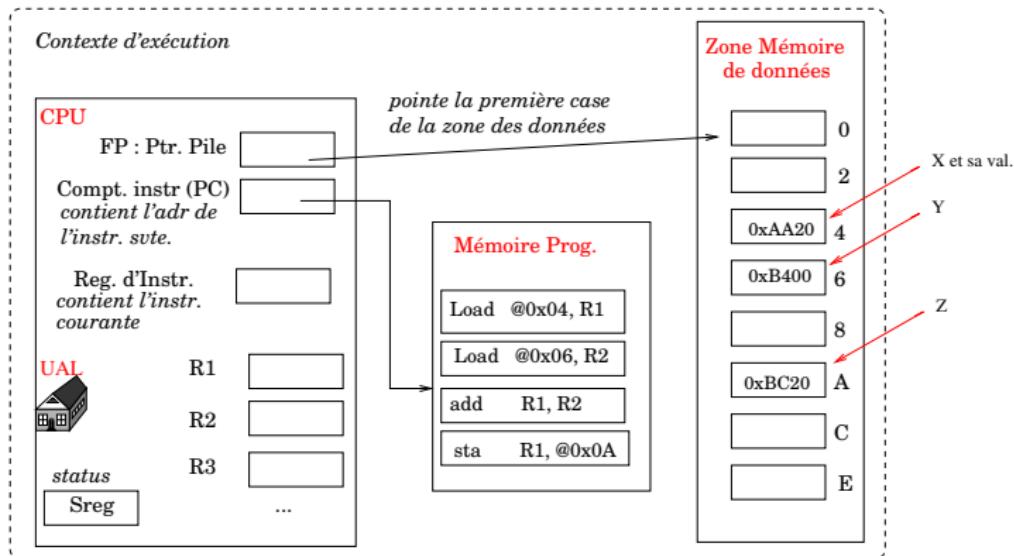
| | | | |
|------|-------|-----------|--|
| 0000 | move | #0, BP+16 | % mettre 0 dans N (déplacement +16 pour N) |
| 0002 | load | BP+16, R | % charger N dans le registre R |
| 0004 | add | #1, R | % ajouter la constante 1 à R |
| 0006 | store | R, BP+14 | % sauvegarde R dans M (déplacement +14 pour M) |
| 0008 | push | BP+14 | % mettre M au sommet de la pile |
| 0010 | call | affiche | % appeler la fonction affiche |
| 0012 | ret | | % retour à l'appelant |
| 0014 | data | 4 | % places pour les variables M et N |

- Si l'adresse de début = X, on ajoute X à toutes les adresses (registre BP/FP/etc.).
 - BP pour le début du code, FP pour le début des données ;
 - Cette *translation* peut être dynamique (en cas de *re-location*)
- A propos des appels de fonctions (utilisation de la pile des appels).
 - Nbr. paramètres ajouté à la pile (ou codé dans l'appelée)

Addendum : Anatomie du contexte d'un processus

Exemple sur un processeur un peu plus récent (avec des registres R_i).

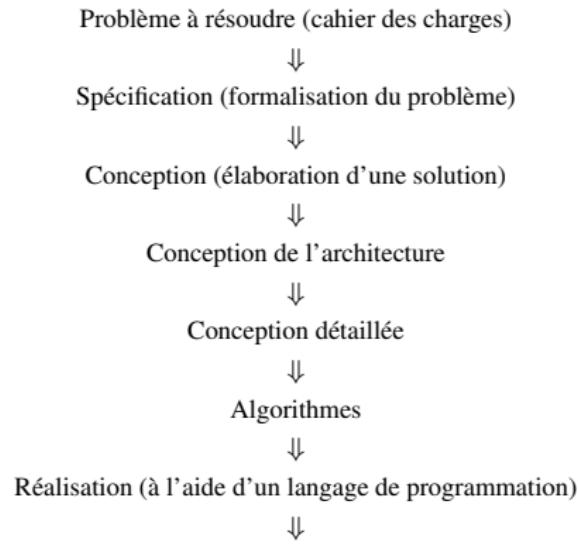
$$Z \leftarrow X + Y$$



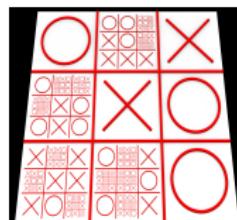
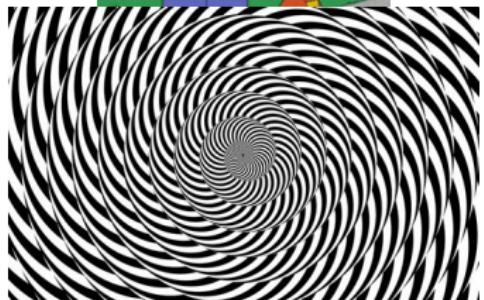
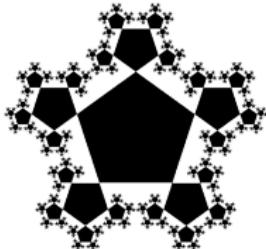
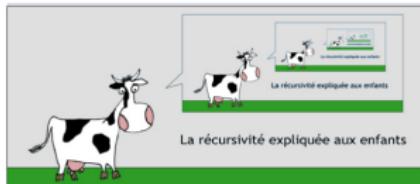
- Convention : **@k** veut dire **FP+k** avec FP = début de données (on peut omettre FP)

Addendum : Place et rôle de l'algorithmique

- Méthode de développement des applications (logiciels)
- Maîtrise de la complexité du développement d'une application
- **Cycle de vie d'Algorithmes :**



Récursivité et Récurrence

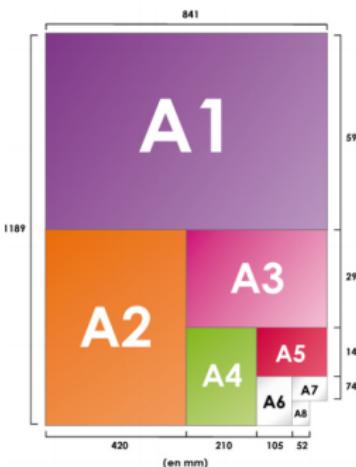


Récursivité et Récurrence (suite)

Le format A4 (*Thanks to Didier Müller, www.nymphomath.ch*)

- Le format A0 (selon la norme DIN-A) est le plus grand format normalisé (1 mètre carré de surface) et se décline jusqu'au format A10.
- La longueur du format inférieur est systématiquement égale à la largeur du format supérieur.
 - Le format inférieur s'obtient en pliant le format supérieur en deux dans sa largeur.

- Quel que soit le format, on trouve toujours le rapport $\sqrt{2}$ entre longueur et largeur.
- A_i est obtenu de A_{i-1} en pliant dans sa longueur la feuille de papier.
- **La relation de récurrence** est donc :
 - Longueur de A_i = Largeur de A_{i-1}
 - Largeur de A_i = $\frac{1}{2}$ Longueur de A_{i-1}



Récursivité : exemples

- Puissance ($\log_2(n) + 1$ appels au lieu de : $x \times x \times x \times \dots \times x$)

```
def puissance(x, n):
    if n==0: return 1
    elif n==1: return x
    elif n%2==0: return puissance(x*x, n//2)
    else: return puissance(x*x, n//2)*x
```

- Factorielle

```
def factorielle(n):
    if n==1 or n==0 : return 1
    else : return n*factorielle(n-1)
```

→ La version itérative :

```
def factorielle(n):
    res = 1
    for i in range(1,n+1): res = res * i
    return res
```

- ☞ Pour $n > 10$, on préfère en général utiliser la formule de *Stirling* :

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \quad \text{avec } e = 2.718281828459, \quad \text{taux d'erreur de calcul : 0.008}$$

Récursivité : exemples (suite)

- Coefficients binomiaux :

de la formule de *Pascal* : $C(n + 1, p + 1) = C(n, p) + C(n, p + 1)$

```
def coefbin(n,p):
    if (n>p) and (p>0):
        return coefbin(n-1,p-1)+coefbin(n-1,p)
    return 1
```

- ☛ Que penser de la fonction suivante :

```
def vis(x) :
    if vis(x) == 1 : return 1
    else : return 0
```

- Condition de terminaison de la récursivité.

Récursivité : exemples (suite)

- Nombre Romain vers Décimal :

Le pseudo-algorithme sera de la forme suivante pour un nombre romain :

- Si la première lettre de ce nombre a une valeur inférieure au deuxième, alors on le soustrait de la valeur de tout le reste.
- Sinon, on l'additionne à la valeur de tout le reste.
- Si le nombre romain a un seul chiffre , prendre simplement la correspondance ($M = 1000, D = 500, \dots$) .

Exemple : soit le tableau **val** : $[M = 1000, D = 500, C = 100, L = 50, X = 10, V = 5, I = 1]$

Détails pour MCMXCIX : on calcule **valeur(MCMXCIX)** .

```

| Lettre = M           (On va de gauche à droite)
| successeur(M) = C et C < M : le résultat final sera val(M) (1000) + valeur(du reste) (de CMXCIX).
| La valeur du reste est la suivante :
| | C < M : valeur(CMXCIX) = valeur(MXCIX) - val(C)
| | La valeur de MXCIX est la suivante :
| | | M > X : val(M) + valeur(XCIX).
| | | valeur(XCIX) = valeur(CIX) - val(X) car le premier X < son successeur C.
| | | valeur(CIX) = 100 + valeur(IX) car C est plus grand que I.
| | | | valeur(IX) = val(X) - val(I) , soit 10 - 1 = 9.          Plus d'appel récursif, on remonte
| | | | valeur(CIX) = val(C) + 9 = 109
| | | valeur(XCIX) = valeur(CIX) - val(X) = 109 - 10 = 99
| | | valeur(MXCIX) = val(M) + valeur(XCIX) = 1000 + 99 = 1099
| | valeur(CMXCIX) = valeur(MXCIX) - val(C) = 1099 - 100 = 999
valeur(MCMXCIX) = 1000 + 999 = 1999

```

- Le nombre MIM (qui pourtant n'existe pas chez les romains) vaut 1999.
- On n'effectue pas ici un test d'exactitude de la chaîne passée en paramètre, on se contentera de l'évaluer.

Remarques sur la récursivité et Python

Quelques arguments en faveur de la représentation récursive :

- Présenter simplement des algorithmes plus naturellement (et plus efficaces).
 - cf. tri rapide (v. + loin)
- Les compilateurs d'aujourd'hui sont plus puissants : ils optimisent très bien un programme présenté de façon abstraite et sans effets de bord (cf. récursivité terminale).
- Structures de données récursives conçues pour leur capacité d'abstraction / efficacité.
 - Difficile d'écrire des algorithmes itératifs sur certaines SDs (cf. Arbres, Graphes).
variable dans une fonction, elle est locale.
- *Nihil Dogma* : utiliser la solution itérative, quand elle existe (cf. *Fib*, *Factorielle*).
- Les algorithmes récursifs nécessitent une pile (d'appels).
- L'espace nécessaire à la sauvegarde des contextes d'un appel récursif au suivant :
observer, réduire

Remarques sur la récursivité et Python (suite)

- Éviter de passer des paramètres dont on n'a pas besoin dans les appels successifs.
 - Pas de passage par référence en Python : on peut les déclarer globales.
 - Ce qui nous met (dans ce cas) à l'abri des variables locales créées et conservées dans les fonctions pendant les appels récursifs (cf. Pile des appels vue plus haut).
- La difficulté d'implanter et l'espace requis pour cette pile ont privés certains langages de la récursivité (cf. Fortran < 90).
- Inefficacité ? oui quand mal utilisée (*le chien mal aimé est accusé de la rage !*).
- Un vieux débat mais les progrès de la compilation des langages de programmation réduit encore la différence en efficacité.
- Il y a aussi les schémas de **transformation**.
- Remarques sur la récursivité **terminale / non-terminale**

Types de données récursifs

Traitement orienté structure de données (*Data-driven algorithms*) :

- Un type récursif est défini en faisant référence textuellement à lui même.
- Les définitions récursives des données est un élément important dans l'analyse et l'écriture d'algorithmes manipulant les données récursives.
- Elles permettent l'emploi plus efficace des méthodes et techniques de programmation dirigée par les données (*Data-Driven*) et *orientée syntaxe*.
- **Exemple 1** : définition du type LISTE

Une LISTE est

soit VIDE

soit un Élément suivi d'une LISTE.

- Cette définition peut être notée par (le formalisme BNF) :

$\langle \text{Liste} \rangle ::= \text{vide} \mid \langle \text{Élément} \rangle \langle \text{Liste} \rangle.$

$\langle \text{Élément} \rangle ::= \text{un élément de type quelconque}$

Types de données récursifs (suite)

- **Exemple 2 :** définition du type CHAINE de caractères

$\langle \text{Chaîne} \rangle ::= \text{vide} \mid \text{caractère } \langle \text{Chaîne} \rangle.$

- La représentation des chaînes est une question d'interprétation ;
sa représentation originelle (*string*) est donnée par une liste/tableau de caractères.

- **Exemple 3 :** définition d'arbre binaire :

$\langle \text{Arbre_bin} \rangle ::= \text{vide} \mid \langle \text{Noeud} \rangle \langle \text{Arbre_bin} \rangle \langle \text{Arbre_bin} \rangle$

$\langle \text{Noeud} \rangle ::= - \text{une donnée} -$

N.B. On voit parfois une définition équivalente :

$\langle \text{Arbre_bin} \rangle ::= \text{vide} \mid \langle \text{Feuille} \rangle \mid$

$\langle \text{Noeud} \rangle \langle \text{Arbre_bin} \rangle \langle \text{Arbre_bin} \rangle$

$\langle \text{Feuille} \rangle ::= - \text{une donnée} -$

Types de données récursifs (suite)

Abstraction récursive (en Python) : fonctions récursives

```
def fonction(x) :
    ...
    if f(x) : ....          # f(x) : condition d'arrêt (de la récursion)
    fonction(ρ(x))        # ρ(x) fait converger x pour que f(x)=True
    ...

```

- Appel récursif à *fonction(.)* dans le texte.
- Faute de convergence de x par $\rho(.)$: risque de non terminaison.
- Le schéma ci-dessus est un schéma de récursivité **Directe**.
- Dans un schéma **indirecte**, *fonction* appellera une autre qui à son tour appellera *fonction* (cf. calcul de *sin* qui appelle *cosin* qui appelle *sin* ...).
- **Exemple 1** : définition de x^y , $y \geq 0$, $x \neq 0$

$$\begin{aligned}x^y &= 1 && \text{si } y = 0 \\&= x * x^{y-1} && \text{si } y > 0\end{aligned}$$

Ici, $\rho(y) : y - 1$

Types de données récursifs (suite)

En Python :

```
def puissance(x,y) :  
    if y==0 : return 1  
    return x * puissance(x,y-1)
```

- **Exemple 2** : définition de $N!$, $N \geq 1$

$$\begin{aligned} N! &= 1 && \text{si } N = 1 \\ &= N.(N-1)! && \text{si } N > 1 \end{aligned}$$

En Python :

```
def fact(N) :  
    if N==1 : return 1  
    return N * fact(N-1)
```

- ☞ En toute rigueur, un entier a une définition récursive (axiomes de Peano)

Types de données récursifs (suite)

- **Exemple 3 :** définition de $\text{maximum}(L)$, $L : \text{liste de taille } \geq 1$

→ Par la définition récursive de la liste L , on a accès à la $\text{tete}(L)$ et $\text{reste}(L)$.

$$\begin{aligned}\text{maximum}(L) &= \text{tete}(L) && \text{si taille}(L) = 1 \\ &= \max(\text{tete}(L), \text{maximum}(\text{reste}(L))) && \text{si taille}(L) > 1\end{aligned}$$

En Python :

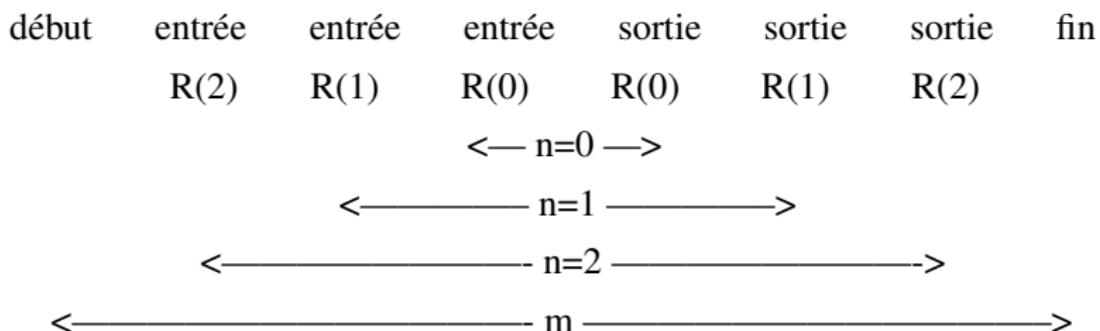
```
# fonction non définie (dite partielle) sur la liste vide
def maximum(L) :
    if len(L)==1 : return L[0]
    return max(L[0], maximum(L[1:]))
```

- La fonction $\max(.,.)$ donne le plus grand parmi ses 2 paramètres.
- Ici, la fonction $\rho(L) \equiv \text{reste}(L)$ écarte la tête de L .
- Le cas $L = []$ non traité (voir les *exceptions*).

Récursivité et durée de vie des variables

- Dans :

```
def R(n) :  
    if n > 0 : R(n-1)  
  
# Appel :  
m=2  
R(m)
```



Récursivité et durée de vie des variables (suite)

- affichage de n : 10 .. 1 (avec décalage dans l'écriture).

Observer la trace du programme Python ci-dessous pour en comprendre le fonctionnement.

```
def écrire_desc(n) :
    if n > 0 :
        print(' '*n, ' n= ', n, " avant l'appel récursif")      # ' '*n : écrit n espaces.
        écrire_desc(n-1)

écrire_desc(10)

"""
TRACE :
    n= 10 avant l'appel récursif
    n= 9 avant l'appel récursif
    n= 8 avant l'appel récursif
    n= 7 avant l'appel récursif
    n= 6 avant l'appel récursif
    n= 5 avant l'appel récursif
    n= 4 avant l'appel récursif
    n= 3 avant l'appel récursif
    n= 2 avant l'appel récursif
    n= 1 avant l'appel récursif
"""

```

Récursivité et durée de vie des variables (suite)

- Observer la trace du programme Python ci-dessous pour en comprendre le fonctionnement.

```
def ecrire_asc(n) :
    if n > 0 :
        ecrire_asc(n-1)
        print(' '*n, 'n= ', n, " après l'appel récursif")

ecrire_asc(10)

"""
TRACE :
n= 1 après l'appel récursif
n= 2 après l'appel récursif
n= 3 après l'appel récursif
n= 4 après l'appel récursif
n= 5 après l'appel récursif
n= 6 après l'appel récursif
n= 7 après l'appel récursif
n= 8 après l'appel récursif
n= 9 après l'appel récursif
n= 10 après l'appel récursif
"""
```

- affichage de n : 1 .. 10 (avec décalage dans l'écriture).
- ☞ Observez le moment de l'appel et son effet sur les valeurs affichées :
- Ici, chaque appel à `print(..)` est suspendue et sera activé au retour de l'appel récursif.

Récursivité et durée de vie des variables (suite)

- Observer les traces du programme Python ci-dessous pour en comprendre le fonctionnement.

```
def desc(n) :
    if n > 0 :
        print(' *n, 'Dans desc( ,n, ), avant l'appel récursif')
        desc(n-1)
        print(' *n, 'Retour dans desc( ,n, )')
    else : print('Appel de desc( ,n, )')
desc(10)
""" TRACE :
    Dans desc( 10 ), avant l'appel récursif
    Dans desc( 9 ), avant l'appel récursif
    Dans desc( 8 ), avant l'appel récursif
    Dans desc( 7 ), avant l'appel récursif
    Dans desc( 6 ), avant l'appel récursif
    Dans desc( 5 ), avant l'appel récursif
    Dans desc( 4 ), avant l'appel récursif
    Dans desc( 3 ), avant l'appel récursif
    Dans desc( 2 ), avant l'appel récursif
    Dans desc( 1 ), avant l'appel récursif
    Appel de desc( 0 )
    Retour dans desc( 1 )
    Retour dans desc( 2 )
    Retour dans desc( 3 )
    Retour dans desc( 4 )
    Retour dans desc( 5 )
    Retour dans desc( 6 )
    Retour dans desc( 7 )
    Retour dans desc( 8 )
    Retour dans desc( 9 )
    Retour dans desc( 10 ) """
```

Analyse et Décomposition récurrente

- Une décomposition récursive est fondée sur la démarche suivante :

Résoudre le problème dans le cas général en se ramenant aux cas particuliers où la solution est naturellement simple.

- Plus précis : pour résoudre un problème, suivons le cheminement suivant :

Trouver au moins un cas où la solution est déjà connue puis tenter de ramener le problème posé pour une valeur quelconque au même problème mais posé sur une valeur plus simple, c'est à dire plus "proche" du cas connu.

- **Exemple** : pour calculer la somme des éléments d'une liste ($somme(L)$) :

Cas connue : $somme(liste_vide) = 0$

Cas récurrent : calculer $somme(reste(L))$ puis y ajouter la valeur de $tete(L)$

- En écartant un par un les éléments de la liste, on se rapproche de la liste vide

→ C'est le *cas connu* : on sait que $somme([]) = 0$

.../..

Analyse et Décomposition récurrente (suite)

- Autrement dit : pour calculer $somme(L)$

Supposons savoir calculer $somme(reste(L))$

Une fois qu'on a cette somme, on y ajoute $tete(L)$

- Et pour $somme(reste(L))$?

Puisque L est une liste, on lui applique le même raisonnement !

- Où cette histoire nous mène-t-elle ? → à la liste vide !

- Peut-on prouver la justesse de la démarche ? → Oui !

→ Bienvenu dans l'*Induction Mathématique*.

Analyse et Décomposition récurrente (suite)

Un exemple magique : trier une liste (méthode Quick-Sort)

- Cas connus = ce que l'on sait et on sait faire :

- Une liste vide est déjà triée
- Une liste avec un élément est déjà triée
- Cela suffit pour trier une liste. **Mais comment ?**

- Pour trier une liste de L

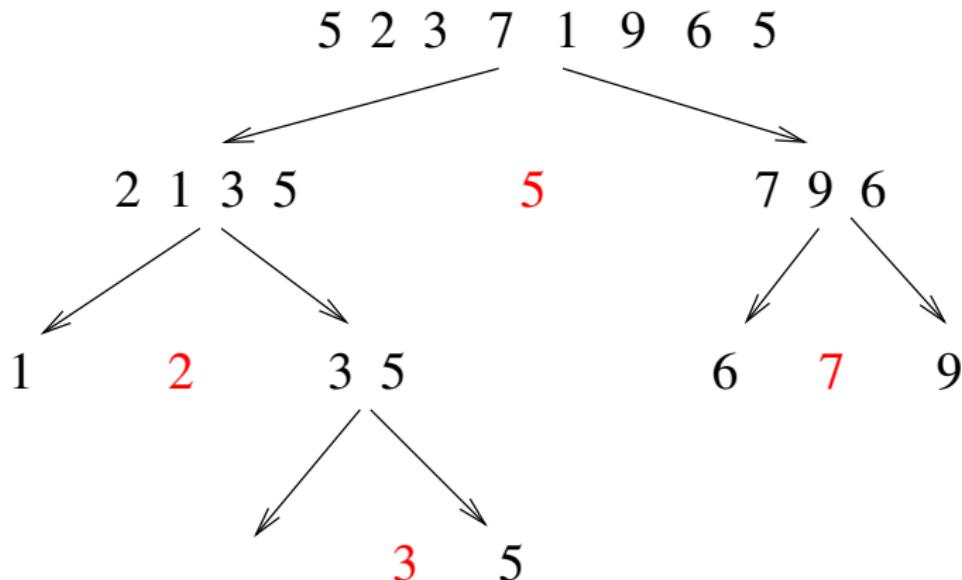
- Si L est de taille ≤ 1 : rien à faire (L déjà triée)
- Choisir un élément de L (en général *tete(L)* appelé *pivot*)
- Scinder L en 2 sous listes L1 et L2 :

L1 : les éléments de L \leq pivot, L2 : ceux $>$ pivot

- Trier L1 et L2 suivant ce même schéma
 - Quand L1 et L2 seront triées, recomposer $L = L1 \oplus [pivot] \oplus L2$ (\oplus : *append*)
- Et hop ! Les éléments sont triés en remontant...

Analyse et Décomposition récurrente (suite)

Exemple de tri : les divisions successives.



Exemple : Médiane

- La méthode suivante est destinée à trouver la médiane d'une séquence.
 - Soit la séquence S . On généralise $k = 1..|S|$ (au lieu de $k = |S|/2$).
 - Pour un nombre v appartenant à S , on peut scinder S en 3 sous tableaux :
 - S_l : contenant les éléments plus petits que v
 - S_v : contenant les éléments $= v$ (contenant v lui-même)
 - S_r : contenant les éléments de S plus grands que v
- Exemple : $S = <2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1>$ et $v=5$:

$$S_l = <2, 4, 1>, S_v = <5, 5>, S_r = <36, 21, 8, 13, 11, 20>$$

- Pour trouver le k ième plus petit élément, il suffit de repérer le sous tableau susceptible de contenir cet élément et de traiter celui-ci.
- Pour S ci-dessus, si $k=8$, on sait que le 8e plus petit élément ne peut être que dans S_r car la somme des tailles de S_l et $S_v=5$.

On notera : $\text{selection}(S, 8) = \text{selection}(S_r, 3)$ sachant la taille de $S_l=3$ et de $S_v=2$.

Exemple : Médiane (suite)

- Généralisation : kème plus petit élément

$$\begin{aligned} \text{selection}(S, k) &= \text{selection}(Sl, k) && \text{si } k \leq |Sl| \\ &= v && \text{si } |Sl| < k \leq |Sl| + |Sv| \\ &= \text{selection}(Sr, k - |Sl| - |Sv|) && \text{si } k > |Sl| + |Sv| \end{aligned}$$

- On répétera récursivement sur le sous tableau concerné jusqu'à arriver à un singleton qui est le résultat recherché.
- Les découpages déséquilibrés peuvent conduire à une complexité défavorable.
- Voir [Aho & al][Wirth] pour 2 algorithmes de complexité moyenne $O(N)$ pour obtenir le kème plus petit élément de S.
→ Comparer aux méthodes qui trient d'abord !

Exemple : Médiane (suite)

Un exemple :

déroulement pour la médiane de $S = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$ avec $|S|=11, k=6$

$\text{selection}(S, 6) : V1=2, Sg1=\langle 1 \rangle, Sv1=\langle 2 \rangle, Sd1=\langle 36, 5, 21, 8, 13, 11, 20, 5, 4 \rangle$

$$k > |Sg1| + |Sv1| \rightarrow \text{selection}(Sd1, 6 - |Sg1| + |Sv1|) = \text{selection}(Sd1, 4)$$

$\text{selection}(Sd1, 4) : V2=36, Sg2=\langle 5, 21, 8, 13, 11, 20, 5, 4 \rangle, Sv2=\langle 36 \rangle, Sd2=\langle \rangle$

$$k < |Sg2| \rightarrow \text{selection}(Sg2, 4)$$

$\text{selection}(Sg2, 4) : V3=5, Sg3=\langle 4 \rangle, Sv3=\langle 5, 5 \rangle, Sd3=\langle 21, 8, 13, 11, 20 \rangle$

$$k > |Sg3| + |Sv3| \rightarrow \text{selection}(Sd3, 4 - |Sg3| + |Sv3|) = \text{selection}(Sd3, 1)$$

$\text{selection}(Sd3, 1) : V4=21, Sg4=\langle 8, 13, 11, 20 \rangle, Sv4=\langle 21 \rangle, Sd4=\langle \rangle$

$$k < |Sg4| \rightarrow \text{selection}(Sg4, 1)$$

$\text{selection}(Sg4, 1) : V5=8, Sg5=\langle \rangle, Sv5=\langle 8 \rangle, Sd5=\langle 13, 11, 20 \rangle$

$$|Sg5| < k \leq |Sg5| + |Sv5| \quad \text{car } 0 < 1 \leq 1$$

→ le résultat est : $V5 = 8$

Il y a bien 5 éléments < 8 et 5 éléments ≥ 8

Exemple : Médiane (suite)

- Solution Python :

```
from random import randint
def mediane(Seq, Seq_Prec, K, K_prec) :
    def scinder(Seq, pivot) :
        Left=[X for X in Seq if X < pivot]
        Eq_pivot=[X for X in Seq if X == pivot]
        Right=[X for X in Seq if X > pivot]
        return (Left, Eq_pivot, Right)

    # ----- Corps de la fonction mediane -----
    print("Seq = ", Seq, " K = ", K)
    if len(Seq)==0 : return -1
    if len(Seq) == len(Seq_Prec) and K != K_prec:
        print('Problème, tous identiques !')
        return Seq[0]

    if K==1 :
        if len(Seq) == 1 :
            print('l\'',K,'ème est ', Seq[0])
            return Seq[0]

    pivot=Seq[0]
    Left, Eq_pivot, Right = [],[],[]
    Left, Eq_pivot, Right=scinder(Seq, pivot)

    if Left==[] and Right==[] :
        return Eq_pivot[K-1]

    print(Left, Eq_pivot, Right)

    Choix=[]
    Val=0
```

Exemple : Médiane (suite)

```
if len(Left) >= K :
    Choix= Left
    Val=K
elif len(Left)+len(Eq_pivot) >= K :
    Choix= Eq_pivot
    Val= K-len(Left)
else :
    Choix = Right
    Val = K - len(Left)-len(Eq_pivot)
return mediane(Choix, Seq, Val, K)

#_____ PP et tests _____
# On teste 100 fois avec vérification automatique

for i in range(100) :
    L=[randint(1,100) for i in range(10)]
    K=5
    print('La liste triée = ', sorted(L), 'on doit trouver ', sorted(L)[K-1])
    Res=mediane(L,L,K,K)
    assert(Res == sorted(L)[K-1])
```

Sortir d'un labyrinthe !

- S'appuie sur les retours arrières
- Utilisé également en DAT (logique, systèmes experts,...)

Prédicat AES_le_premier_succes_suffit;

Données : G : un graphe d'états ;

$Noeud_courant$: le noeud (ou état) courant que l'on traite dans cet appel

Résultat : Succès ou Echec (un booléen)

début

```

    si Noeud_courant est un état final alors
        |
        | renvoyer Succès
    sinon
        |
        pour tous les Noeud_suivant successeurs de Noeud_courant dans G faire
            |
            si Prometteur(G, Noeud_suivant) alors
                |
                | renvoyer Succès
                |
                fin
            fin
        fin
    fin
    renvoyer Echec
fin

```

☞ A propos des heuristiques...

Résumé de la récursivité

- D'une manière générale, la récursivité peut intervenir dans
 - la définition de **structures de données** (chaîne, liste, graphes, arbres,)
 - la définition de **traitements** (algorithmes) : directe et indirecte
- La récursivité s'identifie :
 - lorsque dans la définition d'un objet, un composant fait intervenir l'objet lui même.
 - lorsqu'on appelle une procédure (ou fonction) dans le texte de sa propre définition.
 - cela s'applique à la fois aux données et aux traitements.
- La récursivité est parfois le moyen le plus naturel de spécification (de données et / ou de traitement).
- Bien vérifier qu'une fonction récursive se termine et qu'elle est correcte (preuve ?)

Résumé de la récursivité (suite)

Un autre exemple : définition de lien de parenté (sans alliance) :

X et Y sont parents

- soit Si Y est père, mère, fils ou fille ... de X
- soit s'il existe un individu Z tel que X est parent de Z et Z est parent de Y.

• Comment exprimer la même définition de façon itérative ?

→ Obligerait à recourir à la notion de "chaîne de parenté" (la longueur ?).

• Soit le type de données *Personne*. On utilise le schéma :

$\text{parent}(x, y)$:

$$\forall z \in \text{famille}(x), z \neq x$$

while $\neg\text{Done}$

$$\text{Done} \leftarrow (z = y) \vee \text{parent}(z, y)$$

Done contient le résultat

• En Python :

.../..

Résumé de la récursivité (suite)

```
def parent(x,y) :  
    sont_parents = False  
    for z in famille(x) :           # famille(x) renvoie la famille de x et z != x  
        if z == y or parent(z,y)  
            sont_parents = True  
            break  
    return sont_parents
```

- Une définition récursive = définition par récurrence (e.g. le triangle de Pascal).
- Les objets engendrés par une définition récursive (structures de données ou de calculs) doivent être finis pour permettre la terminaison.
 - En d'autres termes, il faut que des cas triviaux (*connus*) existent.
- Un problème se prête particulièrement bien à l'analyse récursive lorsqu'il peut être décomposé en plusieurs sous-problèmes de même type mais de taille plus petite.

Terminaison : démontrer la convergence (par Induction/Récurrence)

Correction (justesse) : par exemple par l'Induction Mathématique ...

Récurrence et Preuve

Un exemple introductif de preuve :

- Le code python du calcul du reste de la division de deux entiers naturels par soustraction est-il juste ?

```
def reste(a,b) :      # Avec b > 0
    if b <= a : return reste(a-b, b)
    else : return a

reste(13,7)  # 6
reste(3,7)   # 3
reste(9,9)   # 0
```

• **Preuve** :

- pour $a < b$, le reste de la division est (par définition) égal à a :

$$a < b \rightarrow \text{reste}(a, b) = a$$

- pour $b \leq a$, le reste de la division de a par b est le même que celui de la division de $a - b$ par b :

$$b \leq a \rightarrow \text{reste}(a, b) = \text{reste}(a - b, b) \quad \text{CQFD}$$

Preuve par l'Induction Mathématique

- L'induction permet de prouver la justesse d'un raisonnement récursif.
→ L'induction *constructive* permet de trouver un schéma récurrente.

Principe de récurrence (faible)

- Soit P une propriété (ou prédictat) sur \mathbb{N}
 - 1- si $P(0)$ est vérifiée et
 - 2- si, de l'hypothèse " $P(n - 1)$ est vérifiée", on peut déduire la conclusion " $P(n)$ est vérifiée" alors quel que soit l'entier n , $P(n)$ est vérifiée.
- Ce principe peut être étendu :
 - 1- si $\exists n_0 \in \mathbb{N}$ tel que $P(n_0)$ est vérifiée et
 - 2- si de l'hypothèse " $P(n - 1)$ est vérifiée et $(n - 1) \geq n_0$ ", on peut déduire la conclusion " $P(n)$ est vérifiée" alors quel que soit l'entier $n \geq n_0$, $P(n)$ est vérifiée.

Exemple 2

- Prouver que pour tout entier $n \geq 4$, $2^n < n!$.

utiliser pour encadre *Fib* précédent

Étape de base : pour $n = 4$, $2^4 < 4!$ est vrai car $16 < 24$

Étape d'induction :

Soit l'entier $k \geq 4$ et supposons $2^k < k!$ est vraie.

Démontrons que $2^{k+1} < (k+1)!$ doit être vraie :

$$\rightarrow 2^{k+1} = 2 \cdot 2^k < 2 \cdot k! \quad (\text{par l'hypothèse de l'induction})$$

$$\dots < 2 \cdot k! < (k+1) \cdot k! = (k+1)! \quad (\text{car } 2 < k+1)$$

- Détails : on peut montrer que $2 \cdot 2^k < (k+1) \cdot k!$

\rightarrow Ce qui est en rouge vient de l'hypothèse de l'étape k

\rightarrow Évident : sachant que $a \cdot b < c \cdot d$ si $a \leq c$ et $b < d$ pour les entiers $a, b, c, d > 1$,

(Pour notre cas, remplacer $a \cdot b$ par $2 \cdot 2^k$ et $c \cdot d$ par $(k+1) \cdot k!$)

Exemple Postier : raisonnement par cas

Dans un bureau de poste, un postier prétend pouvoir satisfaire toute demande d'achat de timbres de plus de 8 centimes uniquement avec des timbres de 3 et 5 centimes.

→ Prouver que $\forall n \geq 8, n = 3t + 5c$ (t = le nombre des timbres à 3 centimes et c = le nombre de timbres à 5 centimes).

Étape de base : pour $n=8 \rightarrow t=1$ et $c=1$.

Étape d'induction : on montre que pour $k > 8, P(k) \implies P(k+1)$:

Si pour $k \geq 8, P(k) = \text{vrai}$ (avec $k = 3t + 5c$), prouvons alors $P(k+1)$:

- Deux cas peuvent se présenter : $c = 0$ ou $c > 0$ (le nombre de timbres à 5 centimes utilisés pour l'étape k).

$$\textcircled{1} \quad c = 0 \rightarrow k = 3t \wedge t \geq 3 \rightarrow k = 3(t-3) + 9$$

$$\rightarrow k+1 = 3(t-3) + 10 = 3(t-3) + 2 \times 5$$

$$\textcircled{2} \quad c > 0 \rightarrow k = 3t + 5c \wedge c \geq 1 \rightarrow k = 3t + 5(c-1) + 5$$

$$\rightarrow k+1 = 3t + 5(c-1) + 6 = 3(t+2) + 5(c-1)$$

Suite (timbres)

Remarques :

- L'étape d'induction ci-dessus a utilisé une preuve par cas.
- En marge de l'exemple du Postier, considérons l'expression $\forall n \geq 8, n = 3t + 5c$ (t =le nombre des timbres à 3 centimes et c =le nombre de timbres à 5 centimes).
 - Précisons : $n = 3t + 5c, n \geq 8, t \geq 0, c \geq 0, t + c \geq 2$
 - On a $n = 3t + 5c = 3t + 3c + 2c$
 $= 3(t + c) + 2c = 3q + 2c$ (on pose $t + c = q \geq 2$)
 - On peut aussi satisfaire toute demande $n \geq 8$ avec des timbres de 2 et 3 cents.
 - ☞ Si on relâche la condition $q \geq 2$ (la condition > 8 centimes), la formulation du problème change :
 - On pourrait satisfaire toute demande supérieure à **un centime** ! (En exo)

Induction Forte

Principe de récurrence complète (forte)

- Soit P une propriété (prédicat) sur N
 - 1- si $P(0)$ est vérifiée et
 - 2- si de l'hypothèse "pour tout $k < n$, $P(k)$ est vérifiée", on peut déduire la conclusion " $P(n)$ est vérifiée" alors quel que soit l'entier n , $P(n)$ est vérifiée.
- Le cas (1) correspond au *cas de base* : les cas connus du raisonnement récursif
Le (2) correspond à la *récurrence* (donne lieu aux appels récursifs dans le code)

☞ L'importance du *cas de base* (faute de quoi on peut prouver n'importe quoi !)

- Par exemple, si $P(n) : \sum_{i=0}^n 2^i = 2^{n+1} - 1$

On pourra certes démontrer que $P(n-1) \implies P(n)$, $n > 0$;

Pourtant, ni $P(n)$ ni $P(n-1)$ ne sont vrais !

- A propos : $P(n) : \sum_{i=0}^n 2^i = 2^{n+1} - 1$ (le vrai, à démontrer).



Induction Forte (suite)

Exemple : encadrement de Fibonacci

- Montrer que (fonction *Fibonacci*) :

$$\forall n \geq 0, F_n < 2^n \quad \text{pour } n \text{ entier où } F_k = Fib(k).$$

Avec : $Fib(0) = 0$, $Fib(1) = 1$,

et $Fib(n) = Fib(n - 1) + Fib(n - 2)$ pour $n > 1$

Étape de base : $P(0)$ et $P(1)$ sont vrais :

$$\rightarrow F_0 = 0 < 1 = 2^0 \text{ et } F_1 = 1 < 2 = 2^1$$

Étape d'induction (forte) : supposons que $P(k)$ et $P(k + 1)$ sont vrais.

Démontrons que $P(k + 2)$ est vrai :

$$\rightarrow F_{k+2} = F_k + F_{k+1} < 2^k + 2^{k+1} < 2^{k+1} + 2^{k+1} = 2^{k+2}$$

☞ Donne une borne sup. de la complexité de la fonction récursive *Fib(.)*

Induction Constructive

Répond aux questions du genre :

comment trouver que $\sum_1^n k = \frac{n(n+1)}{2}$?

- On pose $S_n = \sum_1^n k = An^2 + Bn + C$
- Sachant que :

$$S_0 = C = 0$$

$$S_1 = A + B + C = 1$$

$$S_2 = 4A + 2B + C = 3$$

- On obtient $\{A = 1/2, B = 1/2, C = 0\}$.

- D'où $S_n = \sum_1^n k = \frac{n^2}{2} + \frac{n}{2} = \frac{n(n+1)}{2}$

☞ Faire la preuve de cette hypothèse pas Induction Mathématique/..

Induction Constructive (suite)

Solution (induction faible) :

Prouver que pour tout entier $n \geq 1$, $1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$.

Étape de base : pour $n=1$, on a $P(1) = \frac{1(1+1)}{2} = 1$, donc vrai

Étape d'induction : Soit l'entier $k \geq 1$ et supposons que $P(k)$ est vrai :

$$1 + 2 + \dots + k = \frac{k(k+1)}{2} \text{ (l'hypothèse de l'induction)}$$

→ On prouve que $P(k+1)$ est vraie, c-à-d. :

$$1 + 2 + \dots + (k+1) = \frac{(k+1)(k+2)}{2}$$

Posons : $1 + 2 + \dots + (k+1) = 1 + 2 + \dots + k + (k+1)$

$$\begin{aligned} &= \frac{k(k+1)}{2} + (k+1) = \frac{k(k+1)}{2} + \frac{2(k+1)}{2} \\ &= \frac{(k+2)(k+1)}{2} \end{aligned}$$

Induction Constructive (suite)

Un autre exemple :

- Comment sait-on que $\text{sum}(1^2 + 2^2 + \dots + n^2) = (n)(n+1)(2n+1)/(6)$?

- Une solution est de penser à la somme comme une intégration.

Sachant $\int x^n dx = x^{(n+1)}/(n+1) + C$

On doit pouvoir exprimer cette somme de manière similaire avec la somme des puissances.

→ Pas toujours simples à intégrer !

- Mais on sait :

la somme(k^2), $k = 1..n$ devrait être cubique mais on ne connaît pas encore ses coeffis.

$$S_n = \sum_1^n k^2 = An^3 + Bn^2 + Cn + D$$

Induction Constructive (suite)

- On a aussi : $S_0 = 0,$

$$S_1 = 1^2 = 1,$$

$$S_2 = S_1 + 2^2 = 5,$$

$$S_3 = S_2 + 3^2 = 14.$$

- On obtient les valeurs $A, B, C, D :$

$$S_0 = D = 0$$

$$S_1 = A + B + C + D = 1$$

$$S_2 = 8A + 4B + 2C + D = 5$$

$$S_3 = 27A + 9B + 3C + D = 14$$

- Avec $\{A = 1/3, B = 1/2, C = 1/6, D = 0\}$, on aura :

$$\begin{aligned} S_n &= \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} \\ &= \frac{n(2n^2 + 3n + 1)}{6} = \frac{n(n+1)(2n+1)}{6} \end{aligned}$$

Induction Constructive (suite)

- **Remarque :** on a laissé tomber

$$\sum_1^n k^2 = \frac{n^3}{3} + (\text{termes de puissance inférieures de } n)$$

C-à-d. différente de $\int_0^N x^2 dx = \frac{N^3}{3} + (\text{termes de puissance inf de } n)$

→ Ces termes sont nuls dans intégrale (d'origine).

- Retour à la somme calculée :

$$S_n = \frac{n(n+1)(2n+1)}{6}$$

- **Cette construction n'est pas une preuve,** c'est une **hypothèse argumentée**.

→ On utilise l'Induction Mathématique pour la prouver. .../..

Induction Constructive (suite)

- Montrer que $p(n) = \sum_1^n i^2 = \frac{n(n+1)(2n+1)}{6}$

Solution :

$$\begin{aligned} \text{D'après } p(n) &= \sum_1^n i^2, \text{ on a } P(n+1) = P(n) + (n+1)^2 \\ &= \frac{n(n+1)(2n+1)}{6} + (n^2 + 2n + 1) = \dots = \frac{(n+1)(2n^2 + 7n + 6)}{6} \end{aligned}$$

$$\begin{aligned} \text{Par ailleurs : } P(n+1) &= \frac{(\cancel{n+1})(\cancel{n+1}+1)(2(\cancel{n+1})+1)}{6} && (\text{si } p(n+1) \text{ était vrai !}) \\ &= \dots = \frac{(\cancel{n+1})(2n^2 + 7n + 6)}{6} \end{aligned}$$

- Dans ce cas, les calculs ont donné des résultats similaires.

☞ La preuve seule $P(n) \implies P(n+1)$ ne suffit pas.

→ Il faut que $P(x_0)$ (ici $x_0 = 1$) soit également prouvé :

→ c'est le cas : $P(1) = \frac{1(1+1)(2 \times 1 + 1)}{6} = 1$ est vérifiable !

Addendum récursivité : Le compte est bon

- Le jeu 'le compte est bon' : atteindre 952 avec les nombres 25, 50, 75, 100, 3, 6.

$$100 + 3 = 103$$

$$103 \times 6 = 618$$

$$618 \times 75 = 46350$$

$$46350 / 50 = 927$$

$$927 + 25 = 952$$

Algorithme de résolution (marche dans 94% des cas) :

1. on sélectionne une paire de nombres
2. on fait une opération sur cette paire
3. si on a trouvé le nombre voulu, STOP
4. on sauvegarde le tableau courant dans un tableau auxiliaire
5. on remplace les deux nombres utilisés par le résultat de l'opération.
On obtient donc un tableau auxiliaire plus petit d'un élément
6. on trie le tableau auxiliaire par ordre décroissant
7. recommencer le raisonnement sur ce tableau (appel récursif)
8. si on n'a pas épuisé les quatre opérations, aller à 2
9. si on n'a pas examiné toutes les paires, aller à 1

- En Python (pour 951) : ../../

Addendum récursivité : Le compte est bon (suite)

```
# "le compte est bon" récursif
def operations(t, max):
    global trouve, t1, signe, objectif
    for i in range(4):
        for j1 in range(max-1):
            for j2 in range(j1+1, max):
                if i==0 : a = t[j1] + t[j2]                                # addition
                elif i==1: a = t[j1] - t[j2]                                # soustraction
                elif i==2: a = t[j1] * t[j2]                                # multiplication
                else: # division (si possible)
                    if t[j1] % t[j2] == 0:
                        a = t[j1] // t[j2]
                    else:
                        a = 0
                if a > 0 :
                    if a == objectif :
                        print(t[j1],signe[i],t[j2], '=',a)
                        trouve = True
                        break
                    t1 = t[:]
                    t1[j1] = a
                    t1[j2] = 0
                    t1.sort()
                    t1.reverse()
                    operations(t1, max-1)
                    if trouve :
                        print(t[j1],signe[i],t[j2], '=',a)
                        break
                if trouve :
                    break
            if trouve :
                break
    if trouve :
        break
```

Addendum récursivité : Le compte est bon (suite)

```
# _____ Test _____
signe = '+-*/'
t1 = [0]*6
trouve = False
objectif = 951
nombres = [100, 75, 50, 25, 6, 3]
print("Objectif", objectif)
print("Tirage", nombres)
print("Lire la solution (si elle existe) de bas en haut")
print()
operations(nombres, 6)
```

• Trace :

```
Objectif 951
Tirage [100, 75, 50, 25, 6, 3]
Lire la solution (si elle existe) de bas en haut

95100 / 100 = 951
1268 * 75 = 95100
1250 + 18 = 1268
6 * 3 = 18
50 * 25 = 1250
```

Addendum : Distribution des primes

Un autre exemple de décomposition.

- On veut distribuer une prime de N francs entre M les employés d'une entreprise.
 - Soit $e_1 \dots e_m$ la série ordonnée qui représente le personnel dans l'ordre de leur ancienneté dont la distribution doit tenir compte : le montant affecté à e_i doit être $\geq e_{i+1}$.
 - Certaines primes peuvent être nulles.
 - Il faut donc décomposer un entier N en M sommants $e_1 \dots e_m$ tels que $e_i \geq e_{i+1}$ sachant que certains e_i peuvent toucher une prime nulle.
-
- Écrire une fonction pour calculer le nombre de manières de répartir N en M sommants.
 - Déterminer le montant affecté à chaque personne.

Solution : soit

Montant : le montant restant,

Nb_empl_restants : le nombre de personnes encore sans prime,

Last_montat : le montant affecté à la dernière personne traitée.

Addendum : Distribution des primes (suite)

Analyse et spécification de la fonction *prime* qui donne le nombre de manières différentes.

prime(Montant_restant, Nb_empl_restants, Last_montat) =

- 0 Si (*Nb_empl_restants < 1*) ou (*Montant_restant < 0*)
- 0 Si (*Nb_empl_restants = 1*) et (*Montant_restant > Last_montat*)
 - on ne peut pas donner une prime $> Last_montat$ au dernier
- 1 Si (*Nb_empl_restants = 1*) et (*Montant_restant \leq Last_montat*)
 - on peut donner *Montant_restant* au dernier employé
- 1 Si (*Nb_empl_restants > 1*) et (*Montant_restant = 0*)
 - on peut donner 0 aux employés du rang *Nb_empl_restants...dernier*
- N* Si (*Nb_empl_restants > 1*) et (*Montant_restant > 0*)
 - où $N \leftarrow \sum_1^K prime(Montant_restant - K, Nb_empl_restants - 1, K),$
 - $K \in 1..min(Last_montat, Montant_restant)$
 - à chaque pas, on peut donner *K* à l'employé du rang *Nb_empl_restants*

Addendum : Distribution des primes (suite)

- Version sans trace et sans le calcul des montant des primes :

```

def prime(Montant_restant, Empl_restants, Last_montant) :
    if (Empl_restants < 1) or (Montant_restant < 0) :
        return 0
    elif Empl_restants == 1 :
        if Montant_restant > Last_montant :
            return 0
        else :
            return 1 # Montant_restant <= Last_montant
    elif Montant_restant == 0 :
        return 1 # Empl_restants > 1
    else :
        N = 0
        Min_ = min(Last_montant, Montant_restant) # min de 2 valeurs
        for K in reversed(range(1, Min_+1)) :
            # K donné à l'employé actuel, on va distribuer le reste
            N += prime(Montant_restant-K, Empl_restants-1, K)
        return N

prime(10,3,10) # On a donné au premier 10, il reste 10 pour 3 employés
# ----> 14 réponses.
# Même chose avec M=5 : on obtient 30 réponses.

```

Addendum : Distribution des primes (suite)

- Une version avec plein de traces :

```

def prime(niv, Montant_restant, Empl_restants, Last_montant=None) :
    """ renvoie un entier = nbr de façons de distribuer Montant_restant """
    if Last_montant==None : Last_montant = Montant_restant*10
    print(' '*niv, 'On doit distribuer ', Montant_restant, ' francs entre ', Empl_restants, \
          ' personnes last distr = ', Last_montant)

    if (Empl_restants < 1) or (Montant_restant < 0) :
        print(' '*niv, '--- peut pas ', 0)
        return 0
    elif Empl_restants == 1 :
        if Montant_restant > Last_montant :
            print(' '*niv, '--- peut pas ', 0)
            return 0
        else : # Montant_restant <= Last_montant
            print(' '*niv, '--- peut 1 façon ', 1)
            return 1
    elif Montant_restant == 0 : # Empl_restants > 1
        print(' '*niv, '--- peut 1 façon (0 à TLM)')
        return 1
    else :
        N = 0
        Min_ = min(Last_montant, Montant_restant) # min de 2 valeurs
        for K in reversed(range(1, Min_+1)) :
            # K donné à l'employé actuel, on va distribuer le reste
            print(' '*niv, 'On donne ', K, " à l'actuel employé et on distribue le reste =", \
                  Montant_restant-K, " entre ", Empl_restants-1, ' employés')
            N += prime(niv+5, Montant_restant-K, Empl_restants-1, K)
            # Optimistion : si on n'a pas pu (0 pour last appel de prim, faire break ?
        print(' '*niv, '--- peut ?, nb facons = ', N)
        return N

```

Addendum : Distribution des primes (suite)

- On teste :

```
#----- Tests -----
Montant_restant=10; Empl_restants=3
print(prime(1, Montant_restant, Empl_restants))
# --> 14 manières (exécutez pour voir les traces)

""" Une partie des traces
On doit distribuer 10 francs entre 3 personnes last distr = 100
On donne 10 à l'actuel employé et on distribue le reste = 0 entre 2 employés
    On doit distribuer 0 francs entre 2 personnes last distr = 10
        <-4- peut 1 façon (0 à TLM)
On donne 9 à l'actuel employé et on distribue le reste = 1 entre 2 employés
    On doit distribuer 1 francs entre 2 personnes last distr = 9
        On donne 1 à l'actuel employé et on distribue le reste = 0 entre 1 employés
            On doit distribuer 0 francs entre 1 personnes last distr = 1
                <-3- peut 1 façon 1
                <- peut ?, nb façons = 1
                ....
Montant_restant=10; Empl_restants=5
print(prime(0, Montant_restant, Empl_restants))
# --> 30 manières
```

- Au départ, le dernier montant distribué n'est pas précisé et on choisit une grande valeur (10 fois la prime).
- Le paramètre supplémentaire *niv* est utilisé pour les traces.

Addendum : Distribution des primes (suite)

- Une version qui construit les montants distribués :

```

def prime(niv, Montant_restant, Empl_restants, L, Last_montant=None) :
    """ renvoie un entier montant de prime """
    global Liste_des_attributions
    if Last_montant==None : Last_montant = Montant_restant*10
    print('*niv, 'On doit distribuer ', Montant_restant, ' francs entre ', Empl_restants, \
    ' personnes last distr = ', Last_montant, 'L= ', L)

    if (Empl_restants < 1) or (Montant_restant < 0) :
        print(' *niv, <-1- peut pas ', 0)
        return (0, [])
    elif Empl_restants == 1 :
        if Montant_restant > Last_montant :
            print('*niv, <-2- peut pas ', 0)
            return (0, [])
        else : # Montant_restant <= Last_montant
            print(' *niv, <-3- peut d\'1 façon, L sera ', L+[Montant_restant])
            Lst += [L+[Montant_restant]]
            return (1, L+[Montant_restant])
    elif Montant_restant == 0 : # Empl_restants > 1
        print(' *niv, <-4- peut 1 façon (0 à TLM), L sera ', L+[0])
        Lst += [L+[0]]
        return (1, L+[0])
    else :
        N = 0
        Min_ = min(Last_montant, Montant_restant) # min de 2 valeurs
        for K in reversed(range(1, Min_+1)) :
            L1=L+[K]
            # K donné à l'employé actuel, on va distribuer le reste
            print(' *niv, 'On donne ', K, " à l'actuel employé et on distribue le reste =',\
            Montant_restant-K, " entre ", Empl_restants-1, ' employés')

```

Addendum : Distribution des primes (suite)

```

N1,L1=prime(niv+5, Montant_restant-K, Empl_restants-1, L1, K)
N += N1

# Optimistion : si on n'a pas pu (0 pour last appel de prim, faire break ?
print(' *niv, <-- peut ?, nb façons = ', N)
return (N, Liste_des_attributions)

```

- On teste : le résultat est de la forme (*Nb_solutions*, *Liste_des_combinaisons*) :

```

Liste_des_attributions=[]
Montant_restant=10; Empl_restants=3 ; Last_montant=5
print(prime(1, Montant_restant, Empl_restants, [], Last_montant))

# (5, [[5, 5, 0], [5, 4, 1], [5, 3, 2], [4, 4, 2], [4, 3, 3]])

Liste_des_attributions=[]
Montant_restant=10; Empl_restants=5 ; Last_montant=50
print(prime(1, Montant_restant, Empl_restants, [], Last_montant))

#(30, [[10, 0], [9, 1, 0], [8, 2, 0], [8, 1, 1, 0], [7, 3, 0], [7, 2, 1, 0], [7, 1, 1, 1, 0], [6, 4, 0],
#[6, 3, 1, 0], [6, 2, 2, 0], [6, 2, 1, 1, 0], [6, 1, 1, 1, 1], [5, 5, 0], [5, 4, 1, 0], [5, 3, 2, 0],
#[5, 3, 1, 1, 0], [5, 2, 2, 1, 0], [5, 2, 1, 1, 1], [4, 4, 2, 0], [4, 4, 1, 1, 0], [4, 3, 3, 0],
#[4, 3, 2, 1, 0], [4, 3, 1, 1, 1], [4, 2, 2, 2, 0], [4, 2, 2, 1, 1], [3, 3, 3, 1, 0], [3, 3, 2, 2, 0],
#[3, 3, 2, 1, 1], [3, 2, 2, 2, 1], [2, 2, 2, 2, 2]])

```

- Exécutez pour voir les détails des traces !

Addendum : Dragon récursif et Tkinter

La courbe du dragon : un autre exemple de Décomposition.

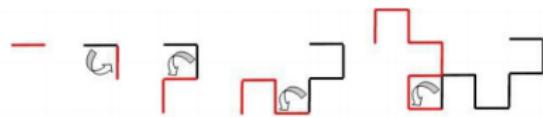
- La courbe du dragon (ou "Fractale du dragon de Heighway") a été pour la première fois étudiée par les physiciens de la NASA John Heighway & al.

La courbe du dragon se construit ainsi :

- Si $t = 0$, l'ordinateur doit dessiner une ligne. C'est la base (ou l'initiateur). La longueur a peu d'importance. On définit la longueur une fois avec s . La commande tortue est : $av(s)$.
- Sinon, si $t > 0$:

$$\text{Dragon}(t) = \text{Dragon}(t - 1) \curvearrowleft \text{Dragon}(t - 1).$$

C'est la règle de récursivité (ou le générateur).



- La machine doit dessiner une courbe avec profondeur de récursion $t - 1$. Ce qui donne :

Dessiner Dragon($t-1$)

Tourner à gauche (90°)

Dessiner Dragon($t-1$)

- Voir quelques détails et remarques en annexes.

Addendum : Dragon récursif et Tkinter (suite)

```
# Courbe du dragon avec instructions à la Logo
from tkinter import *
from math import sin, cos, radians

# _____ commandes _____
def tpos(x0,y0):
    # place la tortue en (x0; y0)
    global x,y
    x = x0
    y = y0

def fcap(angle0):
    global angle
    # oriente la tortue dans une direction (en degrés)
    angle = angle0

def av(d):
    # avance en dessinant
    global x, y
    x2 = x + d*cos(angle)
    y2 = y + d*sin(angle)
    can.create_line(x, y, x2, y2, width=2, fill="black")
    x = x2
    y = y2

def tg(a):
    # tourne à gauche de a degrés
    global angle
    angle -= radians(a)

# _____
```

Addendum : Dragon récursif et Tkinter (suite)

```
def dragon(t, vz):
    if t==0:
        av(15)
    else:
        dragon(t-1,1)
        tg(vz*90)
        dragon(t-1,-1)

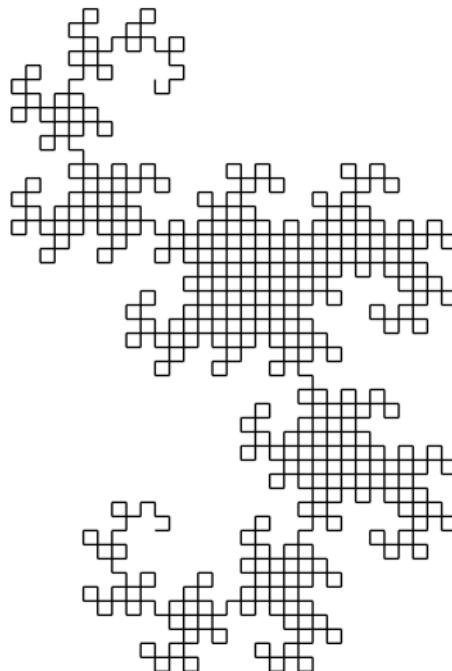
def dessiner():
    fpos(300,600)
    fcap(0)
    dragon(10,1)

#_____  

fen = Tk()
can = Canvas(fen, bg='white', height=800, width=800)
can.pack(side=TOP)

# Boutons
bou1 = Button(fen, text='Quitter', width=10, command=fen.quit)
bou1.pack(side=RIGHT)
dessiner()
fen.mainloop()
fen.destroy()
```

Addendum : Dragon récursif et Tkinter (suite)



Addendum : Un autre ex. d'induction constructive

Le problème d'évaluation de $\sum_{i=1}^n i^3$ en fonction de n . Comment faire ?

- Solution par Induction Constructive :

$$\circ n = 1 : \sum_{i=1}^1 i^3 = 1^3 = 1$$

$$\circ n = 2 : \sum_{i=1}^2 i^3 = 1 + 2^3 = 1 + 8 = 9$$

$$\circ n = 3 : \sum_{i=1}^3 i^3 = 9 + 3^3 = 9 + 27 = 36$$

$$\circ n = 4 : \sum_{i=1}^4 i^3 = 36 + 4^3 = 36 + 64 = 100$$

$$\circ n = 5 : \sum_{i=1}^5 i^3 = 100 + 5^3 = 100 + 125 = 225$$

- Le motif qui apparaît : 1, 9, 36, 100, 225 des carrées.

$$1 = 1^2, 9 = 3^2, 36 = 6^2, 100 = 10^2, 225 = 15^2, \dots$$

Addendum : Un autre ex. d'induction constructive (suite)

Rappel : le motif 1, 9, 36, 100, 225 avec $1 = 1^2$, $9 = 3^2$, $36 = 6^2$, $100 = 10^2$, $225 = 15^2$,

- Les nombres levés au carré diffèrent par n à chaque fois :

- $n = 1 : 1 = 1$
- $n = 2 : 3 = 1 + 2$
- $n = 3 : 6 = 1 + 2 + 3$
- $n = 4 : 10 = 1 + 2 + 3 + 4$
- $n = 5 : 15 = 1 + 2 + 3 + 4 + 5$

- Hypothèse : $\sum_{i=1}^n i^3 = \left(\sum_{i=1}^n i\right)^2$

→ La somme des cubes de n entiers naturels consécutifs=le carré de la somme des n nombres.

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

$$\sum_{i=1}^n i^3 = \left(\frac{n(n+1)}{2}\right)^2 = \frac{n^2(n+1)^2}{4}$$

Addendum : Un autre ex. d'induction constructive (suite)

- Ceci n'est pas une preuve : la formule a été devinée (hypothèse) par les motifs de quelques valeurs de n . Pour en être certain, on peut utiliser l'Induction Mathématique.
- Si l'hypothèse est vérifiée pour un k , est-elle vérifiée pour $k + 1$?

$$\sum_{i=1}^{k+1} i^3 = \frac{(k+1)^2(k+2)^2}{4}$$

→ On ajoute le prochain cube.

$$\begin{aligned} (k+1)^3 + \frac{k^2(k+1)^2}{4} &= (k+1)(k+1)^2 + \frac{k^2(k+1)^2}{4} \\ &= \frac{(k+1)^2(4(k+1) + k^2)}{4} = \frac{(k+1)^2(4k+4+k^2)}{4} = \frac{(k+1)^2(k+2)^2}{4} \end{aligned}$$

- L'hypothèse est vérifiée.

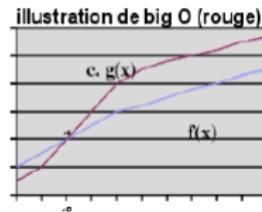
- ☞ L'apprentissage artificielle à partir d'exemples (positifs ou négatifs) est par excellence un cas d'induction constructive : on généralise au fur et à mesure d'exploitation des données, en particulier dans les méthodes incrémentales (e.g. RN, IBL).

Complexité

- Quel est l'intérêt de la complexité (temporelle) des algorithmes ?
- (Familles de) Fonctions de complexité $O(\cdot)$, $o(\cdot)$, $\Omega(\cdot)$, $\omega(\cdot)$, $\Theta(\cdot)$.
- Complexité Worst-Case (cas pire : pessimiste)

Idée informelle : la limite supérieure d'une fonction (modulo un facteur constant)

- Si $g(n)$ est une limite supérieure de $f(n)$, il est alors possible de trouver une valeur (n_0) telle que $f(n) \leq c.g(n)$, pour tout $n \geq n_0$ et c une constante.



Définition : soit f et g deux fonctions de \mathbb{R} dans \mathbb{R} .

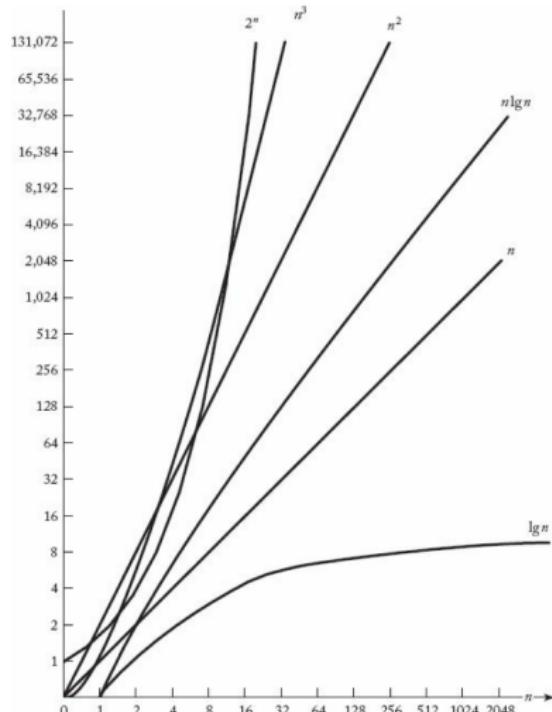
On dit que f est d'ordre inférieur ou égal à g (ou d'ordre au plus g) si l'on peut trouver un réel x_0 et un réel positif c tels que $\forall x \geq x_0$, $f(x) \leq c \cdot g(x)$.

→ g devient plus grand que f à partir d'une certaine valeur x_0 à un facteur c près.

On remarque que des cas d'égalité sont possibles.

Complexité (suite)

- Comparaisons des courbes des croissances usuelles :



Complexité (suite)

Tableau des croissances relatives dans l'ordre croissant (avec exemples) :

| | | |
|------------------|----------------------|--|
| c | <i>constante</i> | → accès à un élément dans un tableau |
| $\log N$ | <i>logarithmique</i> | → couper un ensemble en 2 parties égales, recouper |
| \sqrt{N} | | → utilisation dans le calcul des nombres premiers |
| $\log^2 N$ | <i>log. au carré</i> | → recherche dans un B-arbre, dans une forêt |
| N | <i>linéaire</i> | → parcours linéaire d'un ensemble de données |
| $N \cdot \log N$ | | → couper un ens. en 2 + parcours de chaque partie |
| N^2 | <i>quadratique</i> | → parcourir un ensemble une fois par élément d'un autre ensemble de la même taille (cf. tri bulle) |
| N^3 | <i>cubique</i> | → triple boucle (voir exemple réf. des sommes) |
| 2^N | <i>exponentiel</i> | → générer tous les sous-ens. d'un ens. de données |
| $N!$ | <i>exponentiel</i> | → toutes les permutations d'un ens. (ex. tri bête !) |

Sensibilité à la puissance des machines

Principe d'invariance :

Malgré les différences technologiques, la complexité d'un même algorithme sur 2 machines différentes **ne varie que par un facteur constant**.

Exemple :

Soit T = le temps nécessaire pour exécuter un programme sur une machine M1.

Supposons disposer du même temps T pour exécuter le même programme sur une machine M2 qui est 10 fois plus rapide que M1.

➤ De quel facteur (p/r à M1) peut-on augmenter la taille des données traitées sur M2 ? Comment évolue le temps du calcul ?

Sensibilité à la puissance des machines (suite)

- Soit n la taille des données sur M1, n' celle sur M2 pour la même durée T.

- Pour une complexité linéaire, on a $n' = 10n$ (10 fois plus de données)

- Pour une complexité en n^2 , on a $n'^2 = 10n^2 \rightarrow n' = \sqrt{10}n = 3,16n$

- Pour une complexité en n^3 , on a $n'^3 = 10n^3 \rightarrow n' = 2,15n$

....

- Pour une complexité 2^n , on a $2^{n'} = 10 \cdot 2^n$

$$\text{d'où } n' = n + \log_{10} 10 = n + 3,3$$

→ On peut augmenter seulement par 3 données !!

☞ Constat : les progrès en puissance des machines sont négligeables face aux progrès en algorithmique (qui sont plus rares).

→ Quand il y en a (en algorithmique) : c'est la révolution ! (e.g. Fourier)

Sensibilité à la puissance des machines (suite)

Résumé rapport Temps / Taille des données de l'exemple

| Complexités | 1 | $\log_2(n)$ | n | $n\log_2(n)$ | n^2 | n^3 | 2^n |
|--|----------|---------------------|-------|---------------------|-------------------------|---------|----------|
| Évolution de la taille n quand le temps alloué $t \rightarrow 10t$ | ∞ | n^{10} | $10n$ | $(10-\varepsilon)n$ | $n^{\sqrt{10}} = 3,16n$ | $2,15n$ | $n+3,3$ |
| Évolution du temps t quand la taille des données $n \rightarrow 10n$ | t | $\log(10n) = t+3,3$ | $10t$ | $(10+\varepsilon)t$ | 10^2t | 10^3t | t^{10} |

Détails : la valeur de ε est négligeable devant la croissance de la fonction.

- dans la complexité $n \log(n)$, lorsque $n \rightarrow 10n$, on aura :

$$(10n)\log(10n)=10n[\log(n)+3,3]=10n.\log(n)+33n=n\log(n)[10+33/\log(n)]=n.\log(n)[10+\varepsilon].$$

- dans $\log(n)$, lorsque $t \rightarrow 10t$, on aura : $\log(n) \rightarrow 10 \log(n) = \log(n^{10}) \rightarrow n' = n^{10}$
- dans $n \log(n)$, lorsque $t \rightarrow 10t$, on aura : $n.\log(n) \rightarrow 10n.\log(n) = 10n.\log[10n/10]$
 $= 10n\log(10n) - 10n*3,3 = 10n\log(10n) - \varepsilon(n.\log(n)) = (10-\varepsilon)t$
- le cas de la complexité n^2 a été traité plus haut...

Fonction Oméga

Fonction de complexité optimistes :

La fonction Oméga (Ω) place une borne inférieure asymptotique à la complexité.

Définition de la fonction oméga :

Pour une fonction de complexité $f(x)$, $\Omega(f(x))$ est un ensemble de fonctions de complexité $g(x)$ pour lequel il y a un réel positif c et une constante non négative x_0 tels que $\forall x \geq x_0$, $f(x) \geq c g(x)$.

N.B. : on dira que $f(n) = \Omega(g(n))$ si $g(n) = O(f(n))$

La notation Ω est plus précise mais le big-oh est plus simple à calculer.

→ Ω cherche à donner une meilleure estimation de la complexité.

Fonction Oméga (suite)

- Comparaison des fonctions $O(\cdot)$ et $\Omega(\cdot)$

Exemple : soit à estimer la croissance de la somme

$$f(n) = \sum_{j=0}^n j^2 = 1^2 + 2^2 + 3^2 + \dots + n^2$$

- La figure en face montre :

$$\sum_{j=1}^{n-1} j^2 \leq \int_1^n x^2 dx = \left[\frac{x^3}{3} \right]_1^n = \frac{(n^3 - 1)}{3}$$

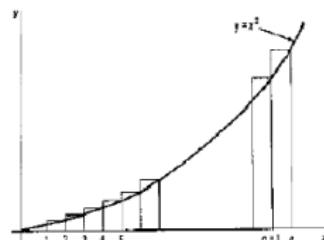
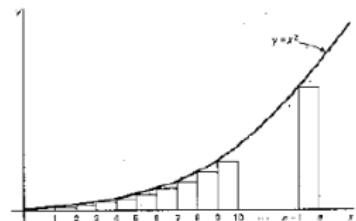
et donc (passer de n à $n+1$)

$$f(n) \leq \int_1^{n+1} x^2 dx \leq \frac{((n+1)^3 - 1)}{3}$$

- De même

$$f(n) = 1^2 + 2^2 + 3^2 + \dots + n^2 \geq \int_0^n x^2 dx = \frac{n^3}{3}$$

$$\text{d'où } \int_0^{n+1} x^2 dx \geq \sum_{i=1}^n i^2 \geq \int_0^n x^2 dx$$



Fonction Oméga (suite)

- Comment trouver une fonction $\theta(\cdot)$? (c-à-d. à la fois $O(\cdot)$ et $\Omega(\cdot)$) :

La figure suivante montre quelques exemples courants de ces 3 fonctions de complexité pour n^2 .

Rappel : les termes d'ordre inférieurs ne sont pas significatifs dans les 3 fonctions.

$$\begin{aligned} & 3\log n + 8 \\ & 5n+7 \\ & 2n \log n \\ & 4n^2 \\ & 6n^2+9 \\ & 5n^2+2n \end{aligned}$$

(a) $O(n^2)$

$$\begin{aligned} & 4n^2 \\ & 6n^2+9 \\ & 5n^2+2n \\ & 4n^3+3n^2 \\ & 6n^6+n^4 \\ & 2^n + 4n \end{aligned}$$

(b) $\Omega(n^2)$

The diagram consists of two overlapping circles. The left circle contains terms from set (a): $3\log n + 8, 5n+7, 2n \log n, 4n^2, 6n^2+9, 5n^2+2n$. The right circle contains terms from set (b): $4n^2, 6n^2+9, 5n^2+2n, 4n^3+3n^2, 6n^6+n^4, 2^n + 4n$. The intersection of the two circles contains terms common to both sets: $4n^2, 6n^2+9, 5n^2+2n$.

(c) $\Theta(n^2) = O(n^2) \cap \Omega(n^2)$

Ordre des fonctions

Les ordres les plus importantes

- L’ordre de croissance de $T(N) \leq f(N)$ → $T(N) = O(f(N))$: *big-Oh*
 - L’ordre de croissance de $T(N) \geq f(N)$ → $T(N) = \Omega(f(N))$: *Omega*
 - L’ordre de croissance de $T(N) \approx f(N)$ → $T(N) = \Theta(f(N))$: *Theta*
 - L’ordre de croissance de $T(N) < f(N)$ → $T(N) = o(f(N))$: *little-Oh*
- *little-oh* n’accepte pas l’égalité (=) alors que *big-oh* l’accepte
- $T(N) = O(f(N))$: $T(N)$ ne va pas croître à une vitesse plus grande que $f(N)$.
→ $f(N)$ est une borne supérieur de $T(N)$.
 - $T(N) = O(f(N)) \rightarrow f(N) = \Omega(T(N))$
→ $T(N)$ est une borne inférieure de $f(N)$.
→ Par exemple, N^3 croît plus vite que N^2 , on peut donc dire que :
$$N^2 = O(N^3) \text{ ou } N^3 = \Omega(N^2).$$

Calcul de complexité

Propriétés des limites de fonctions :

Soit $f(N)$ le temps d'exécution (éventuellement empirique) observé d'un programme et $g(N)$ une famille de fonctions.

- On peut calculer le taux relatif de croissance de 2 fonctions f et g en calculant

$$\lim_{n \rightarrow \infty} \frac{f(N)}{g(N)}$$

Si f et g admettent des limites, on a alors les propriétés suivantes :

- Si $\lim_{n \rightarrow \infty} \frac{f}{g} = c > 0$, f et g sont du même ordre, $f=O(g)$ et $g=O(f) \Rightarrow f = \Theta(g)$.
- Si $\lim_{n \rightarrow \infty} \frac{f}{g} = 0$, alors $f = o(g)$ et f est d'ordre inférieur à g .
- Si $\lim_{n \rightarrow \infty} \frac{f}{g} = +\infty$, f est d'ordre supérieur à g .
 \Rightarrow On note $f = \Omega(g)$ qui est équivalent à $g=o(f)$

- On pourra (également) utiliser la règle "Hôpital" :

si f et g sont dérivables et $\lim f(x) = \lim g(x) = \infty$

alors $\lim_{n \rightarrow \infty} \frac{f(N)}{g(N)} = \lim_{n \rightarrow \infty} \frac{f'(N)}{g'(N)}$ si la limite existe.

Calcul empirique de complexité par programme

Montrer que la probabilité pour que deux entiers distincts et aléatoires $I, J \leq N$ soient premiers entre eux est approche $\frac{6}{\pi^2} = 0.608$ (pour N grand).

Le code suivant calcule le nombre de fois où i et j sont premiers entre eux.

```
def proba_prime(N) :
    Rel=0
    Total_essais=0
    for i in range(1,N):
        for j in range(i+1,N) :
            Total_essais +=1
            if (pgcd(i,j) == 1) : Rel += 1
    return Rel/Total_essais
```

*# Rel : nombre de fois où i et j sont premiers
Total : nombre d'essais
Simulation du tirage aléatoire sans remise
On sait : pgcd est $O(\log N)$*

Analyse de la complexité du code : 2 boucles et $pgcd(O(\log N))$ dans la boucle interne

→ On aura une complexité de l'ordre de $O(N^2 \log(N))$

N.B. : le $pgcd$ est en fait $O(\log(\min(i,j)))$, voire $o(\log(\min(i,j)))$.

→ $\log(N)$ représente le nombre de bits pour représenter l'entier N .

- Dans le tableau suivant, chaque colonne représente $T(N)/g(N)$ et on étudie la limite. .../..

Calcul empirique de complexité par programme (suite)

Le tableau obtenu pour quelques fonctions usuelles de complexité :

| Taille | \ln | $\ln^* \ln$ | N | $N \ln$ | $N+N \ln$ | N^N | $N^* N^* N$ |
|--------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|
| 50 | 0.0000702961 | 0.0000179692 | 0.0000055000 | 0.0000014059 | 0.0000011197 | 0.0000001100 | 0.0000000022 |
| 75 | 0.0001528667 | 0.0000354064 | 0.0000088000 | 0.0000020382 | 0.0000016549 | 0.0000001173 | 0.0000000016 |
| 112 | 0.0003401506 | 0.0000720887 | 0.0000143304 | 0.0000030371 | 0.0000025060 | 0.0000001279 | 0.0000000011 |
| 168 | 0.0008050408 | 0.0001571129 | 0.0000245536 | 0.0000047919 | 0.0000040094 | 0.0000001462 | 0.0000000009 |
| 252 | 0.0016952926 | 0.0003065945 | 0.0000371984 | 0.0000067274 | 0.0000056970 | 0.0000001476 | 0.0000000006 |
| 378 | 0.0175406665 | 0.0029555146 | 0.0002754021 | 0.0000464039 | 0.0000397125 | 0.0000007286 | 0.0000000019 |
| 567 | 0.0194570992 | 0.0030687692 | 0.0002175750 | 0.0000343159 | 0.0000296409 | 0.0000003837 | 0.0000000007 |
| 8500 | 0.7441067491 | 0.0822415378 | 0.0007920841 | 0.0000875420 | 0.0000788294 | 0.0000000932 | 0.0000000000 |
| 1275 | 0.0227007100 | 0.0031746130 | 0.0001273145 | 0.0000178045 | 0.0000156201 | 0.0000000999 | 0.0000000001 |
| 1912 | 0.0419171490 | 0.0055476013 | 0.0001656496 | 0.0000219232 | 0.0000193608 | 0.0000000866 | 0.0000000000 |
| 2868 | 0.0851187651 | 0.0106914718 | 0.0002362838 | 0.0000296788 | 0.0000263669 | 0.0000000824 | 0.0000000000 |
| 4302 | 0.1815649459 | 0.0217005521 | 0.0003531204 | 0.0000422048 | 0.0000376990 | 0.0000000821 | 0.0000000000 |
| 6453 | 0.3971829320 | 0.0452769414 | 0.0005399362 | 0.0000615501 | 0.0000552517 | 0.0000000837 | 0.0000000000 |
| 9679 | 0.8774637252 | 0.0956080934 | 0.0008320189 | 0.0000906564 | 0.0000817491 | 0.0000000860 | 0.0000000000 |
| 14518 | 1.9480021329 | 0.2032737924 | 0.0012858511 | 0.0001341784 | 0.0001214999 | 0.0000000886 | 0.0000000000 |
| 21777 | 4.4913626210 | 0.4496484276 | 0.0020600849 | 0.0002062434 | 0.0001874746 | 0.0000000946 | 0.0000000000 |

- Rappel : la vitesse de la machine n'a pas d'effet sur les rapports $T(N)/g(N)$ et les constantes ('c' de la définition de big-Oh) sont négligées.
- On remarque que la colonne N^2 avec un écart type minimal est un bon candidat pour la fonction de complexité Θ .

Complément à la complexité

- Différentes fonctions de complexité principales (pour toute familles dont Θ) :

- Soit les catégories de complexité suivantes (avec $k > j > 2$ et $b > a > 1$) :

$$\Theta(\log n) \prec \Theta(n) \prec \Theta(n \log n) \prec \Theta(n^2) \prec \Theta(n^j) \prec \Theta(n^k) \prec \Theta(a^n) \prec \Theta(b^n) \prec \Theta(n!)$$

- Si une fonction de complexité $g(n)$ est dans une catégorie qui est à gauche de celle contenant $f(n)$, alors $g(n) \in o(f(n))$ (strictement inférieur)
- On constate que **toute complexité logarithmique est meilleure** que les polynomiales, et les polynomiales sont meilleures que les exponentielles, et les exponentielles sont éventuellement meilleures que factorielles.

Par exemple : $\log n \in o(n)$, $n^{10} \in o(2^n)$, $2^n \in o(n!)$

- Pour $c >= 0$, $d > 0$,

Si $g(n) \in O(f(n))$ et $h(n) \in \Theta(f(n))$

alors $c \cdot g(n) + d \cdot h(n) \in \Theta(f(n))$

Complément à la complexité (suite)

☞ La complexité moyenne \neq la moyenne de complexités

- Un exemple de complexité moyenne :

La Recherche de X dans S[1..n]

- La probabilité pour que $S[k]=X$, $1 \leq k \leq n$ est $1/n$,
- Pour arriver à l'indice $k=1..n$, on aura fait les comparaisons :

$$A(n) = \sum_{k=1}^n \left(k \cdot \frac{1}{n} \right) = \frac{1}{n} \cdot \sum_{k=1}^n k = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2} \quad \text{opérations de base si } X \text{ est dans } S$$

Mais pour être complet, il faut tenir compte du cas où X n'est pas dans S :

- La probabilité pour que X soit dans S = p et pour que $S[k]=X$ est p/n
- La probabilité pour que X ne soit pas dans S = $1-p$
- On fait k comparaisons si $S[k]=X$
et n comparaisons si X n'est pas dans S,

Complément à la complexité (suite)

La complexité moyenne :

$$A(n) = \sum_{k=1}^n \left(k \cdot \frac{p}{n} \right) + n(1-p) = \frac{p}{n} \cdot \frac{n(n+1)}{2} + n(1-p) = n\left(1 - \frac{p}{2}\right) + \frac{p}{2}$$

- si $p=1$, $A(n) = (n+1)/2$
- si $p= \frac{1}{2}$, $A(n) = 3n/4 + \frac{1}{4}$ \Rightarrow seul $\frac{3}{4}$ de S est recherché en moyenne

Remarques sur $A(n)$:

On a supposé une **probabilité identique** pour chaque élément du tableau.

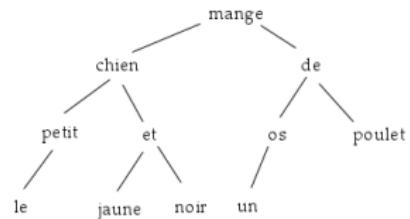
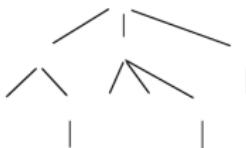
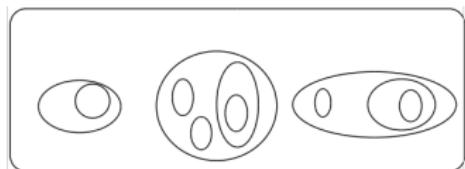
\Rightarrow A changer si l'on connaît une distribution différente de taille/ valeurs :

\hookrightarrow telle valeur est plus fréquente que telle autre...

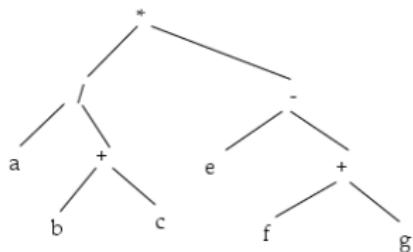
\Rightarrow Le calcul de la complexité moyenne est souvent difficile

Mais donne une indication intéressante (si possible à calculer).

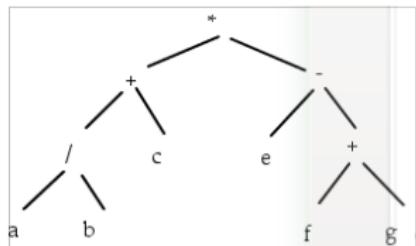
Les arbres



$$(a/(b+c) * (e-(f+g))$$



$$(a/b+c) * (e-(f+g))$$



Les arbres (suite)

Quelques définitions

- arbre, sous-arbre, noeud, racine
- descendant (fils, fille), ascendant (père, mère)
- feuille, mot des feuilles,
- hauteur (niveau)
- arbre n-aire, binaire, ternaire, quaternaire, ...
- Relation d'ordre, ABOH, TAS (Heap), AVL, etc

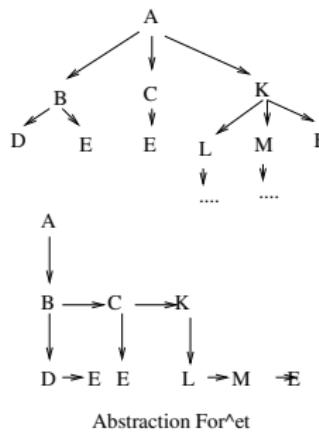
Définition récursive d'un arbre binaire :

Un arbre est

- soit vide
- soit un élément , un arbre (gauche) et un arbre (droit)

Arbres : représentation

- Représentation des arbres par : matrice, liste , table (tableau), ...
- Exemple : représentation d'un arbre **ternaire** par matrices / tableaux / listes
 - Abstraction **Forêt** (d'arbres) : notions de Fils et Frère (utilisation pour les dicos)



| | |
|-----|-------|
| A | B C K |
| B | D E |
| C | E |
| D | A |
| E | |
| K | L M E |
| L | |
| M | |
| ... | |

Représentation par Listes

☞ L'abstraction Forêt permet en fait de présenter un arbre n-aire par un binaire.

Représentation d'Arbres binaires

Une représentation par tableaux (listes) :

Convention :

Si l'indice (Place) du noeud est = i

Alors la Place désignant son fils gauche est à $2i$ et

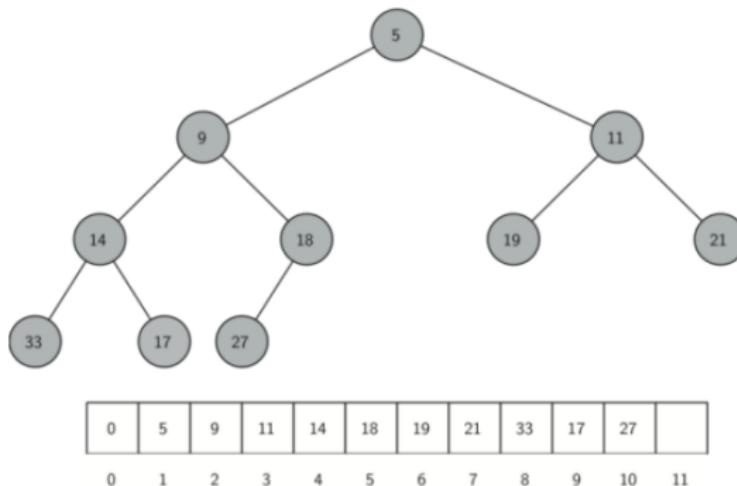
la Place désignant son fils droit est à $2i + 1$.

- La *hauteur* d'un arbre (binaire) h
 - Arbre binaire **équilibré**
 - La hauteur h d'un arbre **équilibré (compacte)** est telle que
$$h = \lfloor \log(NB_ele) \rfloor + 1$$
 d'où $NB_ele = 2^h - 1$
- ☞ **Important** : question d'équilibre de cette représentation.
- ☞ Techniques employées pour équilibrer les arbres, les **AVL** (v. plus loin)

Représentation d'Arbres binaires (suite)

Un exemple de cette représentation :

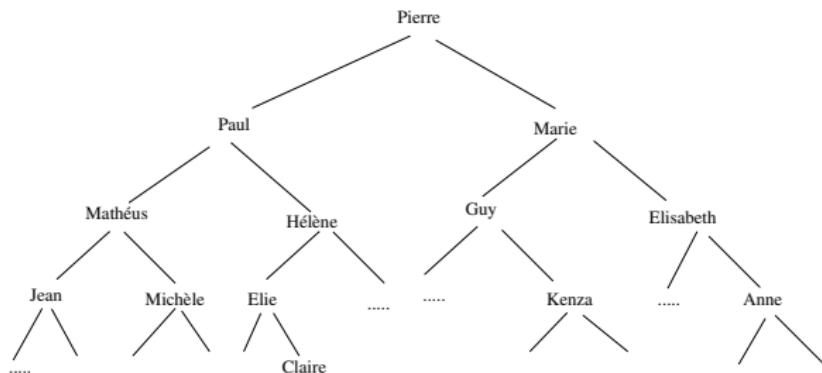
Un arbre binaire *complet* et sa représentation par table / liste



- ☞ Relation d'ordre : on est en présence d'un *Heap* minimal (v. *Heap* plus loin)
- ☞ Problème de compacité : pas trous entre les éléments !

Représentation d'Arbres binaires (suite)

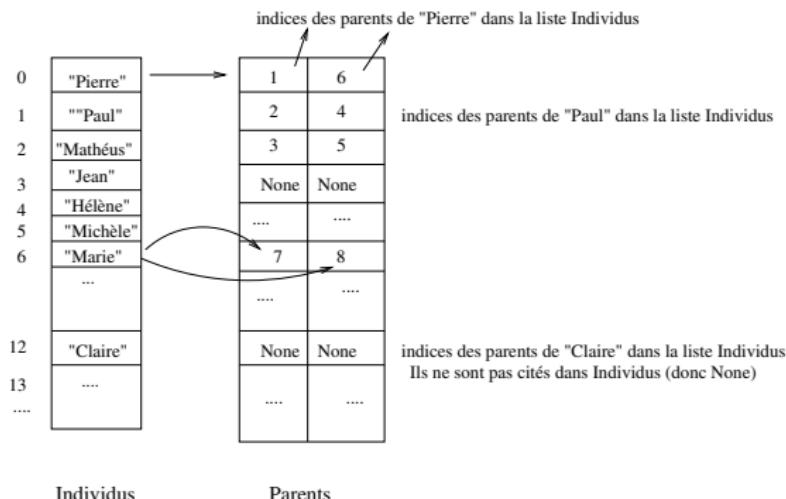
Une autre représentation possible : exemple *généalogie*



- Exemples de questions posées (les développer sur la représentation concrète) :
 - Trouver les ancêtres de (p. ex.) Hélène.
 - Insérer un nouvel enfant ;
 - Afficher l'arborescence, etc...

Représentation d'Arbres binaires (suite)

Représentation par listes :



☞ Comparaison des deux représentations.

→ La 2nde est plus élaborée (temps d'accès plus long) mais plus efficace.

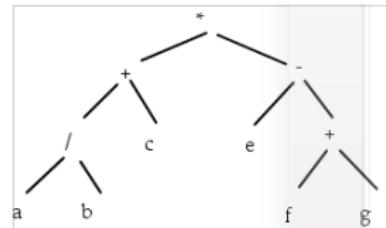
Parcours d'arbre binaire

- Soit : **R** = Racine, **D** = sous-arbre Droit et **G** = sous-arbre Gauche
- Il y a trois types (principaux) de parcours : observer la place de **R**
 - Préfixé : **R G D** : pré-ordre
 - Infixé : **G R D** : mi-ordre
 - Postfixé : **G D R** : post-ordre

Exemples :

Les noeuds visités selon le mode de parcours :

- Infixé : $a/b+c^*e-f+g \rightarrow ((a/b)+c) * (e-(f+g))$
- Postfixé : $a\ b\ / c\ + e\ f\ g\ +\ -\ *$
- Préfixé : $*\ +\ /\ a\ b\ c\ -\ e\ +\ f\ g$



On remarque que lors d'une évaluation (manuelle) de l'expression, seul le mode *infixé* nécessite des parenthèses pour lever des ambiguïtés éventuelles.

Parcours d'arbre binaire (suite)

Les algorithmes de parcours d'arbre binaire en Python :

```
def Prefixe(R) :
    if not vide(R) :
        traiter(R);                      # R
        Prefixe(gauche(R));              # G
        Prefixe (droit(R));             # D

def Infixe (R) :
    if not vide(R) :
        Infixe (gauche(R));            # G
        traiter(R);                   # R
        Infixe (droit(R));            # D

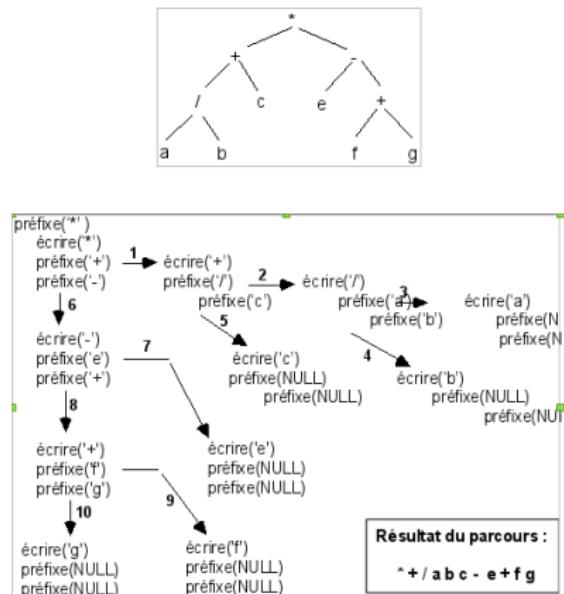
def Postfixe(R) :
    if not vide(R) :
        Postfixe(gauche(R));          # G
        Postfixe (droit(R));         # D
        traiter(R);                  # R
```

- ☞ Le paramètre R représente initialement la racine puis différents noeuds aux cours des appels.
 - Voir plus loin le codage complet en Python.

N.B. : Le TDA est un outil apprécié pour la définition algébrique et formelle de la structure arbre (voir plus loin en Annexes).

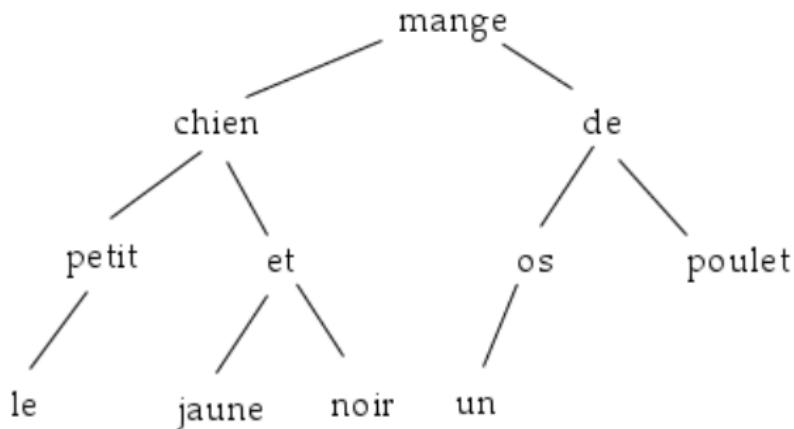
Parcours d'arbre binaire (suite)

Trace du parcours préfixé (pré-ordre) de l'arbre d'une expression :



Parcours d'arbre binaire (suite)

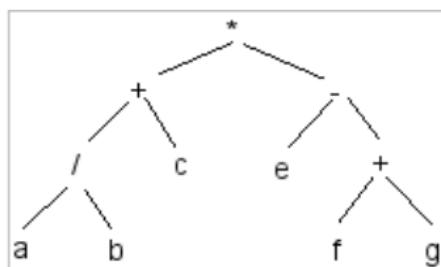
Question : quel type de parcours pour retrouver les informations dans l'ordre dans l'arbre binaire suivant ?



Arbres binaires : un exemple Python

Représentation et parcours Python de l'arbre de l'expression arithmétique précédente :

- On représente cet arbre par une structure similaire à l'exemple *généalogie* ci-dessus.



| indice | Noeuds | Fils |
|--------|--------|--------------|
| 0 | '*' | (1,6) |
| 1 | '+' | (2,5) |
| 2 | '/' | (3,4) |
| 3 | 'a' | (None, None) |
| 4 | 'b' | (None, None) |
| 5 | 'c' | (None, None) |
| 6 | '-' | (7,8) |
| 7 | 'e' | (None, None) |
| 8 | '+' | (9,10) |
| 9 | 'f' | (None, None) |
| 10 | 'g' | (None, None) |

- ☞ Dans le code python qui suit, l'arbre sera une donnée globale.

Arbres binaires : un exemple Python (suite)

```
# Les fonctions recoivent un indice
# La racine tjs en 0

def prefixe(S) :
    global Arbre
    if S != None :
        traiter(S)
        prefixe(gauche(S))
        prefixe(droite(S))

def infixe(S) :
    global Arbre
    if S != None :
        infixe(gauche(S))
        traiter(S)
        infixe(droite(S))

def postfixe(S) :
    global Arbre
    if S != None :
        postfixe(gauche(S))
        postfixe(droite(S))
        traiter(S)
```

Arbres binaires : un exemple Python (suite)

```
# N'est appelé que si S existe (un indice)
def traiteur(S) :
    global Arbre
    print(Arbre[S], end=' ')

# N'est appelé que si S existe (un indice)
def gauche(S) :
    global Fils
    return Fils[S][0]

# N'est appelé que si S existe (un indice)
def droite(S) :
    global Fils
    return Fils[S][1]

# Liste principale + matrice de G/D
def go():
    global Arbre
    global Fils

    Arbre = ['*', '+', '/', 'a', 'b', 'c', '−', 'e', '+', 'f', 'g']

    Fils = [(1,6),          # 0
             (2,5),          # 1
             (3,4),          # 2
             (None, None),   # 3
             (None, None),   # 4
             (None, None),   # 5
             (7,8),          # 6
             (None, None),   # 7
             (9,10),         # 8
             (None, None),   # 9
             (None, None),   # 10
             ]
```

Arbres binaires : un exemple Python (suite)

```
infixe(0); print()
prefixe(0); print()
postfixe(0); print()

#-----#
#           TEST (rapide) : sans construire __main__
#-----#

go()

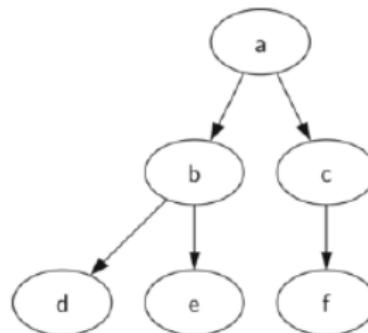
# Trace :
a / b + c * e - f + g
* + / a b c - e + f g
a b / c + e f g + - *
```

- On reprendra cette expression plus loin pour une **évaluation** arithmétique.
- Donner une solution pour la représentation $2i, 2i+1, 2i+2$ (indices : $0..$)
 - Idée d'équilibrage de la représentation $2i, 2i+1, 2i+2$

Arbres : représentation par listes imbriquées

- Une liste de listes contient l'arbre.

```
un_arbre=
  [ 'a',
    [ 'b',
      [ 'd',
        [], []           # racine
      ],
      []
    ],
    [ 'e',
      [], []           # a-b-d
    ],
    []
  ],
  [ 'c',
    [ 'f',
      [], []           # 2 ss-arbres vides de 'd'
    ],
    []
  ]
]                                     # 2 ss-arbres vides de 'e'
]                                     # fin de la partie gauche
]                                     # a-c
]                                     # a-c-f
]                                     # 2 ss-arbres vides de 'f'
]
]                                     # fils droit de 'c'
```



- La racine de l'arbre est accessible par `un_arbre[0]`,
- Le fils gauche par `un_arbre[1]` et le fils droit par `un_arbre[2]`.

Arbres : représentation par listes imbriquées (suite)

- La même convention s'applique aux sous-arbres.

```
un_arbre[0]
# 'a'

un_arbre[1]
# ['b', ['d', [], []], ['e', [], []]]

un_arbre[2]
# ['c', ['f', [], []], []]

un_arbre[1][0]
# 'b'
```

- ☞ Un arbre n-aire peut adapter cette même représentation.
- La définition récursive des listes s'applique aux différentes représentations des arbres où les sous-arbres ont la même structure que l'arbre entier.
- Rappel : pour un arbre binaire, cette définition est :
 - soit *Vide*
 - soit *une_information, Arbre, Arbre*
 - C-à-d. une information puis un sous-arbre (gauche) puis un sous-arbre (droit).

Python : Fonctions usuelles sur arbres binaires

Quelques fonctions usuelles pour les arbres binaires repr. par **listes imbriquées** :

```
def arbre_binaire(r):    return [r, [], []]

def get_val_racine(arbre):
    assert(arbre !=[])
    return arbre[0]

def set_val_racine(arbre ,new_val):
    assert(arbre !=[])
    arbre[0] = new_val

def get_fils_gauche(arbre):
    assert(arbre !=[])
    return arbre[1]

def get_fils_droit(arbre):
    assert(arbre !=[])
    return arbre[2]

def inserer_gauche(arbre , new_gauche):
    def ins_gauche(A,new_gch) :
        if not A[1]  :  A[1]=[new_gch,[],[]]
        else :  ins_gauche(A[1], new_gch)

    assert(len(arbre)>1)
    ins_gauche(arbre , new_gauche)
    return arbre

def inserer_droit(arbre , new_droite):
    def ins_droit(A,new_dt) :
        if not A[2]  :  A[2]=[new_dt,[],[]]
        else :  ins_droit(A[2], new_dt)
    assert(len(arbre)>1)
    ins_droit(arbre , new_droite)
    return arbre
```

Python : Fonctions usuelles sur arbres binaires (suite)

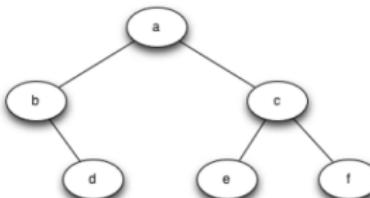
Quelques tests (dessiner l'arbre !) :

```
r = arbre_binaire(3)
print(r)
# [3, [], []]
inserer_gauche(r, 4)
print(r)
# [3, [4, [], []], []]
inserer_gauche(r, 5)
print(r)
# [3, [4, [5, [], []], []], []]
inserer_droit(r, 6)
print(r)
# [3, [4, [5, [], []], []], [6, [], []]]
inserer_droit(r, 7)
print(r)
# [3, [4, [5, [], []], []], [6, [], [7, [], []]]]
l = get_fils_gauche(r)
print(l)
# [4, [5, [], []], []]
set_val_racine(l, 9)
print(l)
# [9, [5, [], []], []]
inserer_gauche(l, 11)
print(l)
# [9, [5, [11, [], []], []], []]
print(r)
# [3, [9, [5, [11, [], []], []], []], [6, [], [7, [], []]]]
print(get_fils_droit(get_fils_droit(r)))
#[7, [], []]
```

Python : Fonctions usuelles sur arbres binaires (suite)

Un autre exemple :

construction de l'arbre ci-dessous à l'aide des opérateurs précédents :



```
a = arbre_binaire('a')
b= inserer_gauche(a,'b') # la valeur renvoyé est l'arbre complet (a)
print(a)
# ['a', ['b', [], []], []]

print(b) # Affichera l'arbre a car b    récupère la totalité de l'arbre
# ['a', ['b', [], []], []]

inserer_droit(a,'c')
inserer_droit(get_fils_gauche(a), 'd')
c=get_fils_droit(a)
inserer_gauche(c,'e')
inserer_droit(get_fils_droit(a), 'f')

print(a)
# ['a', ['b', [], ['d', [], []]], ['c', ['e', [], []], ['f', [], []]]]
```

Python : Fonctions usuelles sur arbres binaires (suite)

Un autre exemple :

afficher (dessiner) l'arbre précédent à l'écran.

- Chaque information (noeud ou feuille) sera affichée sur une ligne.
- L'idée est de descendre le plus à droite possible dans l'arbre pour afficher la feuille la plus à droite sur une ligne puis écrire l'information du parent de cette feuille-là sur la ligne suivante avant d'aller à gauche.

```
def affiche_arbre(arbre) :
    def affiche_arbre_bis(arbre, decalage) :
        if arbre != []:
            affiche_arbre_bis(get_fils_droit(arbre), decalage+3)
            print('*' * decalage, arbre[0])
            affiche_arbre_bis(get_fils_gauche(arbre), decalage+3)
    affiche_arbre_bis(arbre, 1)
```

→ Noter que `affiche_arbre(arbre)` joue le rôle du (*wrapper*) et prépare le paramètre `decalage` (espaces à gauche) pour `affiche_arbre_bis(arbre, decalage)`.

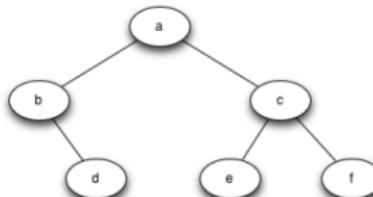
Python : Fonctions usuelles sur arbres binaires (suite)

Trace pour l'arbre **a** ci-dessus (tourner la tête 90 degrés à gauche !) :

```
a=.....
affiche_arbre(a)
```

On obtient :

```
    f
   c
   e
 a
   d
  b
```



Quelques autres algorithmes en Python sur arbres

Quelques algorithmes sur les arbres binaires :

- ① La fonction taille qui calcule le nombre de noeuds d'un arbre binaire.
- ② La fonction feuille qui test si un noeud d'un arbre binaire est une feuille.
- ③ La fonction nb_feuilles qui calcule le nombre de feuilles d'un arbre binaire.
- ④ La fonction hauteur qui calcule la hauteur d'un arbre binaire.

NB : La hauteur est définie par le maximum de nombre d'arcs allant de la racine jusqu'à toute feuille + 1.

- ⑤ La fonction recherche qui cherche l'élément X dans un arbre binaire.
- ⑥ La fonction *isomorphe* qui vérifie si deux arbres contiennent les mêmes informations dans le même ordre.

Quelques autres algorithmes en Python sur arbres (suite)

☞ Les algorithmes suivant font abstraction de la représentation de l'arbre.

- La fonction *taille* qui calcule le nombre de noeuds d'un arbre binaire.

```
def taille(R) :  
    if vide(R) : return 0  
    else : return 1+taille(gauche(R))+taille(droit(R));
```

→ Quel est le type du parcours ?

- La fonction *feuille* : teste si un noeud d'un arbre binaire est une feuille.

```
def feuille(R) :  
    if vide(R) : return False;  
    else : return vide(gauche(R)) and vide((droit(R));
```

→ Quel est le type du parcours ?

Quelques autres algorithmes en Python sur arbres (suite)

- La fonction *nb_feuilles* : calcule le nombre de feuilles d'un arbre binaire.

```
def nb_feuilles(R) :  
    if vide(R) : return 0  
    if feuille(R) : return 1  
    else : return 1+nb_feuilles(gauche(R))+nb_feuilles(droit(R));
```

→ Quel est le type du parcours ?

- La fonction hauteur qui calcule la hauteur d'un arbre binaire.

NB : La hauteur est définie par le maximum de nombre d'arcs allant de la racine (jusqu'à toute feuille) + 1.

```
def hauteur(R) :  
    if vide(R) : return 0  
    else : return max(hauteur(gauche(R)),hauteur(droit(R)))+1
```

→ Avec *taille* et *hauteur*, on peut vérifier la *compacité*

→ Ensuite, soit on ré-équilibre ; soit on utilise les AVL (v. + loin).

Quelques autres algorithmes en Python sur arbres (suite)

- La fonction *recherche* : cherche l'élément X dans un arbre binaire.

```
def recherche(R, Val) :  
    if vide(R) : return False  
    elif info(R) == val : return True  
    else : return recherche(gauche(R)) or recherche(droit(R))
```

→ Quel est le type du parcours ?

- La fonction *isomorphe* : vérifie si deux arbres contiennent les mêmes informations dans le même ordre.

```
def isomorphe(R1, R2) :  
    if vide(R1) and vide(R2): return True  
    elif not vide(R1) and not vide(R2):  
        return info(R1) == info(R2) and isomorphe(gauche(R1),gauche(R2)) and  
               isomorphe(droit(R1),droit(R2))  
    else : return False
```

ABOH en Python

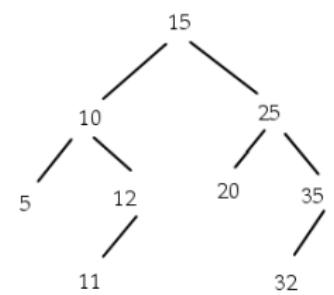
Variante d'arbre : Arbre Binaire Ordonné horizontalement (ABOH)

→ Appelé également Arbre Binaire de Recherche (ABR).

- Relation vérifiée sur chaque noeud d'un ABOH :

$$\text{Max}(\text{info}(ss_arbre_G)) < \text{info}(R) < \text{Min}(\text{info}(ss_arbre_D))$$

- Exemple :



☞ Où se trouve l'élément minimum / maximum (dans l'arbre) ?

ABOH en Python (suite)

Quelques algorithmes sur les ABOH :

- La fonction de recherche d'un élément X dans un ABOH :

```
def recherche_ABOH(R, Val) :  
    if vide(R) : return False  
    elif info(R) == val : return True  
    elif info(R) > val : return recherche_ABOH(gauche(R))  
    else : return recherche_ABOH(droit(R))
```

- Complexité : si h = la hauteur de l'arbre alors

$h = \lfloor \log(N) \rfloor + 1$ (N = nombre de noeuds) si **arbre équilibré et compacte.**

- La version itérative (transformation aisée de la récursivité) :

```
def recherche_ABOH_iter(R, Val) :  
    trouve=False  
    while not vide(R) and not trouve :  
        if info(R) == val : trouve=True  
        elif info(R) > val : R=gauche(R)  
        else : R=droit(R)  
    return trouve
```

ABOH en Python (suite)

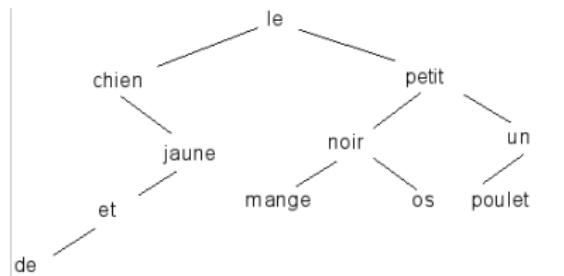
- Les ABOHs sont rarement utilisés pour des insertions.

→ Néanmoins, l'insertion doit être prévue :

```
def insere_ABOH(R, Val) :          # R est un objet mutable
    if vide(R) : return ajouter(R, Val)      # ajout du noeud (Val, vide, vide)
    elif info(R) > val : return insere_ABOH(gauche(R), Val)
    else : return insere_ABOH(droit(R), Val)
```

→ D'autres versions (renvoyer un arbre).

- Exemple : voici l'ABOH obtenu en insérant les mots de la phrase suivante : "le petit chien jaune et noir mange un os de poulet" (utilité ?)



Exercice sur les Arbres

Exercice sur les ABOHs :

- Proposer une représentation d'ABOH sous Python
 - Par une liste principale + liste des fils
 - Par un tableau (liste) et la relation $i, 2i, 2i + 1$
- Lire un texte et constituer l'indexe des mots (comme à la fin d'un livre).

Il est besoin de construire un dictionnaire où chaque lettre de l'alphabet est une entrée pour les mots commençant par celle-ci.

Sur les arbres en général :

- Reprendre l'expression arithmétique vue plus haut (arbres binaires) pour une **évaluation** arithmétique.
- Donner une relation (telle que $2i, 2i + 1, 2i + 2$) pour un arbre ternaire, quaternaire et quintenaire.
 - Pour un arbre quaternaire, voir BE2.

Pile et File en Python

- Voir TAS (plus loin) pour les classes utilisées.

Pile :

- La classe *Queue* implante une (sous-classe) *Pile* (gestion *LIFO*) :

```
class Queue.LifoQueue(maxsize=0)
```

```
import queue
pile = queue.LifoQueue()
for i in range(5): pile.put(i)
while not pile.empty() : print(pile.get(), end=' ')
print()
# Donne 4 3 2 1 0
```

- L'élément inséré en dernier est en tête : gestion LIFO
- La classe *Queue* propose en fait plusieurs structures de données (avec des applications en programmation concurrente).

Pile et File en Python (suite)

File (d'attente) : gestion FIFO

- La classe *Queue* implante également une (sous-classe) *file* :

```
class Queue.Queue(maxsize=0)
```

☞ Rappelons qu'une Pile est facilement réalisable avec une simple liste !

- Opérateurs habituels (communs aux *Queue* ou *LifoQueue*) avec notation Objets :

- *empty()* : test de vacuité
- *full()* : plein (si on a donné une taille max à la création)
- *get()* : récupère l'élément et l'enlève de la Pile/File ; exception *queue.Empty* si vide
- *put()* : ajout (selon *Queue* ou *LifoQueue*) ; exception *queue.Full* si plein (voir *full()*)
- *qsize()* : nombre actuel d'éléments.

Pile et File en Python (suite)

- ☞ La classe file est utilisable en particulier dans les Threads (structure *Thread-Safe*). Il y a de nombreuses utilisations de cette structure en programmation concurrente.
- Exemple d'utilisation d'une file :

```
import queue
file = queue.Queue()
for i in range(5) : file.put(i)
while not file.empty() : print(file.get(), end=' ')
print()
# 0 1 2 3 4
```

- Premier arrivé, premier servi : gestion FIFO
- ☞ Noter *Queue.Queue()* pour une *File* et *Queue.LifoQueue()* pour une *Pile*.

Autres implantations de Pile et File en Python

Pile (LIFO) :

- Certains pensent que sous Python, le type Pile (avec le schéma *last-in, first-out*) est "superflu" puisque une liste remplit les mêmes fonctions :

```
stack = []          # fonction d'initialisation
stack = [3, 4, 5]
stack.append(6)     # fonction empiler(6)
stack.append(7)

stack
# [3, 4, 5, 6, 7]

stack.pop()         # fonction dépiler() qui renvoie sommet(). Attention au TDA
# 7

stack
# [3, 4, 5, 6]

stack[-1]           # fonction sommet(). La pile n'est pas modifiée
# 5
```

Autres implantations de Pile et File en Python (suite)

File (FIFO) :

- Pour les *Files* (d'attente, avec le schéma *First-in, First-out*), on peut utiliser : les listes (comme pour les Piles) :

```
"""Module de gestion d'une queue FIFO."""
queue = [] # initialisation

def enqueue():
    queue.append(int(input("Entrez un entier : ")))

def dequeue():
    if len(queue) == 0:
        print("\nImpossible : la queue est vide !")
    else:
        print("\nElément %d supprimé" % queue.pop(0))

def afficheQueue():
    print("\nqueue :", queue)
```

Autres implantations de Pile et File en Python (suite)

Mieux ? :

- Le type **Deque** de *collections.deque* "double ended queue" (liste à 2 entrées) :

```
from collections import deque
queue = deque(["Eric", "Jean", "Michael"])
queue.append("Pierre")           # enfiler("Pierre")
queue.append("Marie")            # enfiler("Marie")

queue.popleft()                 # défiler() qui renvoie sommet(). Attention différent du TDA
# 'Eric'

queue.popleft()                 # défiler() qui renvoie sommet(). Attention au TDA
# 'Jean'

queue                           # Ce qui reste
# deque(['Michael', 'Pierre', 'Marie'])
```

Autres implantations de Pile et File en Python (suite)

- Les fonctions de *deque* (extrait de la documentation Python) :

append(x) : Add x to the right side of the deque.

appendleft(x) : Add x to the left side of the deque.

clear() : Remove all elements from the deque leaving it with length 0.

copy() : Create a shallow copy of the deque.

count(x) : Count the number of deque elements equal to x.

extend(iterable) : Extend the right side of the deque by appending elements from the iterable argument.

extendleft(iterable) : Extend the left side of the deque by appending elements from iterable.

Note, the series of left appends results in reversing the order of elements in the iterable argument.

index(x[, start[, stop]]) : Return the position of x in the deque (at or after index start and before index stop).

Returns the first match or raises ValueError if not found.

insert(i, x) : Insert x into the deque at position i.

pop() : Remove and return an element from the right side of the deque. If no elements are present, raises an IndexError.

popleft() : Remove and return an element from the left side of the deque.

If no elements are present, raises an IndexError.

remove(value) : Remove the first occurrence of value. If not found, raises a ValueError.

reverse() : Reverse the elements of the deque in-place and then return None.

rotate(n) : Rotate the deque n steps to the right. If n is negative, rotate to the left.

Rotating one step to the right is equivalent to : *d.appendleft(d.pop())*.

Autres implantations de Pile et File en Python (suite)

Deque objects also provide one read-only attribute :

maxlen : Maximum size of a deque or None if unbounded.

- In addition to the above, deques support iteration, pickling, len(d), reversed(d), copy.copy(d), copy.deepcopy(d), membership testing with the in operator, and subscript references such as d[-1].
- Indexed access is O(1) at both ends but slows to O(n) in the middle. For fast random access, use lists instead.
- Starting in version 3.5, deques support __add__(), __mul__(), and __imul__().

Autres implantations de Pile et File en Python (suite)

Exemple :

```
from collections import deque
d = deque('ghi')                                # création
for elem in d:                                  # Itération
    print(elem.upper())
""" G H I """
d.append('j')                                    # Ajout à Droite (= à la fin par défaut)
d.appendleft('f')                               # Ajout à Gauche (utile pour une insertion en tête d'une File/Pile)
d                                         # montrer la représentation de d
# deque(['f', 'g', 'h', 'i', 'j'])

d.pop()                                         # enlever et renvoyer l'élément le plus à droite
# 'j'
d.popleft()                                     # enlever et renvoyer l'élément le plus à gauche
# 'f'

list(d)                                         # présenter "d" comme une liste
# ['g', 'h', 'i']
d[0]                                           # l'élément le plus à gauche
# 'g'
d[-1]                                         # l'élément le plus à droite
# 'i'

list(reversed(d))                            # présenter "d" comme une liste mais à l'envers !
# ['i', 'h', 'g']
'h' in d                                       # recherche dans "d"
# True

d.extend('jkl')                                # ajouter plusieurs éléments en même temps
d
# deque(['g', 'h', 'i', 'j', 'k', 'l'])
```

Autres implantations de Pile et File en Python (suite)

```
d.rotate(1)                      # rotation à droite
d
# deque(['l', 'g', 'h', 'i', 'j', 'k'])
d.rotate(-1)                      # rotation à gauche
d
# deque(['g', 'h', 'i', 'j', 'k', 'l'])

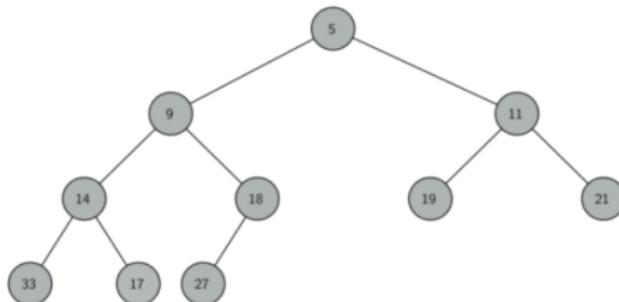
deque(reversed(d))               # créer un nouveau deque avec "d" à l'envers
# deque(['l', 'k', 'j', 'i', 'h', 'g'])

d.clear()                         # vider "d"
d.pop()                           # On ne peut pas enlever un élément de "d" vide
# IndexError: pop from an empty deque

d.extendleft('abc')               # extendleft() met à l'envers l'ordre des éléments ajoutés
d
# deque(['c', 'b', 'a'])
```

TAS ou Heap

- Un TAS (HEAP) est un arbre binaire ordonné **verticalement**.
- Dans un tel arbre, la relation d'ordre est (cf. *min_heap*) :
 $\text{info}(noeud) \leq \min \text{info}(gauche)$ et $\text{info}(noeud) \leq \min \text{info}(droit)$
- Pour le *max_heap*, on aura symétriquement
 $\text{info}(noeud) \geq \max \text{info}(gauche)$ et $\text{info}(noeud) > \max \text{info}(droit)$
- Par défaut, on considère les *min_heaps* comme dans figure suivante :



TAS ou Heap (suite)

- Un *min_heap* binaire (*binary heap*) est également appelé une *file binaire de priorités* (*binary priority queue*).
 - Une file binaire de priorités est (en général) implantée par un TAS.
- L'appellation *file de priorité* vient du fait que les informations stockées représentent (habituellement) des priorités de traitement (comme devant un guichet)
- Un TAS est en général représenté en mémoire par un tableau avec les indices $2i$ et $2i + 1$ (i doit commencer à 1).
- Un TAS peut être implantée à l'aide des listes triées.

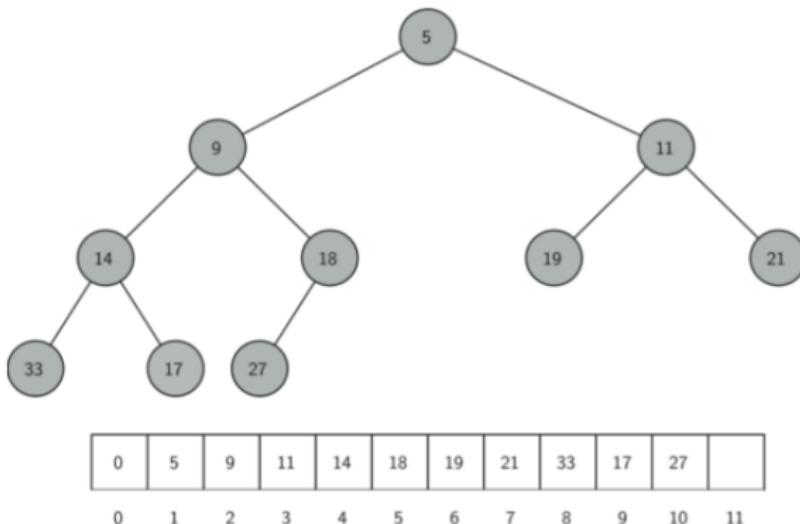
Dans ce cas, l'insertion sera $O(N)$ et le tri $O(N \cdot \log(N))$.

Par contre, dans les Tas binaires (qui sont par définition ordonnés), l'insertion et la suppression (enfiler / défiler) sont $O(\log(N))$.

TAS ou Heap (suite)

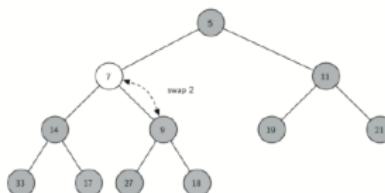
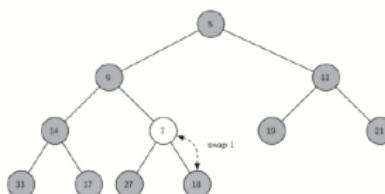
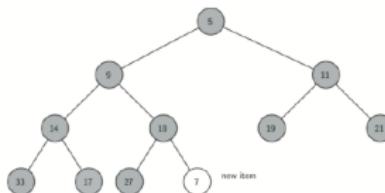
Heap et sa représentation par tableau (liste) :

- Un arbre binaire complet et sa représentation par table / liste



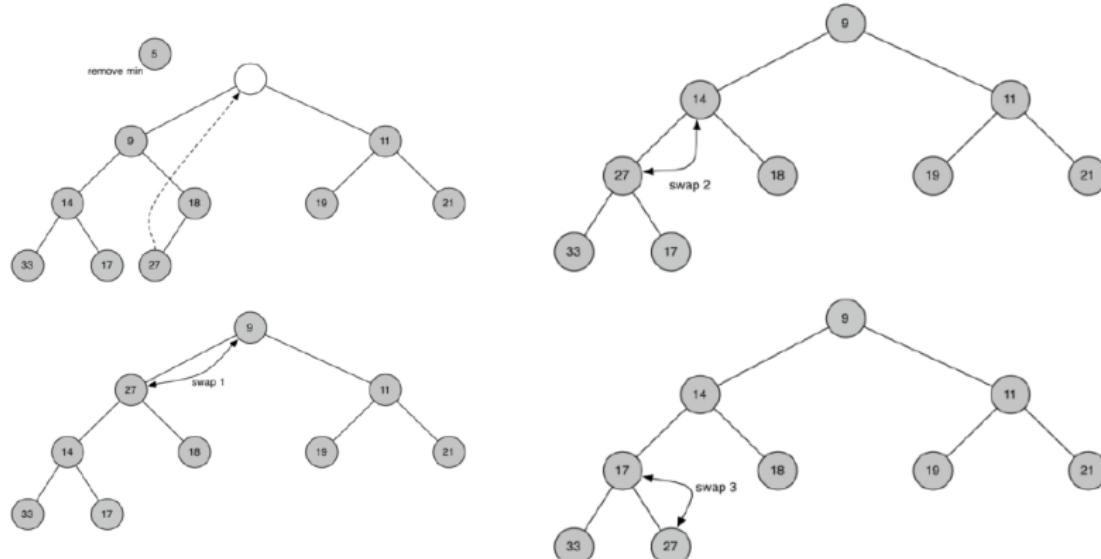
TAS ou Heap (suite)

- Ex d'insertion de 7



TAS ou Heap (suite)

- Exemple du retrait du min et la réorganisation :



TAS en Python

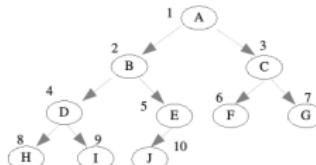
TAS (ou *Heap Queue*) en Python : une variante d'une File de priorité.

- Le module *heapq* implante la méthode *min-heap-sort* pour le tri des listes Python.
- Un TAS est un arbre binaire avec la relation d'ordre $<$ (**minimum** au sommet).
- Son implantation utilise un tableau/une liste (soit *heap*) où $\forall k \geq 0$:

$$\text{heap}[k] \leq \text{heap}[2k + 1] \quad \text{et} \quad \text{heap}[k] \leq \text{heap}[2k + 2].$$

☞ Dans Python, l'élément d'indice 0 existe ! (voir son implantation).

- Les éléments absents sont considérés comme infinis.
- Dans un tel arbre, le minimum est toujours au sommet.



| | | | | | | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|---|---|---|----|-----|----|----|----|
| indice | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | .. |
| valeur | * | A | B | C | D | E | F | G | H | I | J | ... | | | |

TAS en Python (suite)

Exemples : on peut transformer une liste L en $heapq$ par $heapq.heapify(L)$.

Par exemple :

```
import heapq

heap = []
heapq.heappush(heap, (1, 'one'))
heapq.heappush(heap, (10, 'ten'))
heapq.heappush(heap, (5, 'five'))

for x in heap:
    print(x)

print()
# (1, 'one')
# (10, 'ten')
# (5, 'five')

heapq.heappop(heap)      # Enlever le minimum
# (1, 'one')

for x in heap:
    print(x)
print()

#(5, 'five')
# (10, 'ten')

print(heap[0])           # Le minimum actuel
# (5, 'five')
```

TAS en Python (suite)

- Le même exemple modifié : remarquer la destruction du TAS (par un ajout sauvage !) puis sa reconstruction :

```
import heapq

heap = [(1, 'one'), (10, 'ten'), (5, 'five')]
heapq.heapify(heap)
for x in heap:
    print(x)

# (1, 'one')
# (10, 'ten')
# (5, 'five')

heap[0] = (9, 'nine')           # On modifie le 2e élément
for x in heap:                 # ZZ : CE N'EST PLUS UN TAS
    print(x)

# (9, 'nine')
# (10, 'ten')
# (5, 'five')

heapq.heapify(heap)            # ZZ : CE REDEVIENT UN TAS
for x in heap:
    print(x)

# (5, 'five')
# (10, 'ten')
# (9, 'nine')
```

TAS en Python (suite)

- La différence d'un *heapq* par rapport à une File de priorité est
 - *heapq* est (seulement) un *min-heap*
 - Toute intervention (sauvage) sur le tableau sous-jacent risque de détruire le TAS
→ Mais on peut réparer (cf. l'exemple ci-dessus).
- Exemple avec *Heapq* (de la bibliothèque) : min_heap et max_heap :

```
import heapq
listForTree = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
heapq.heapify(listForTree)           # Pour min_heap (par défaut)
heapq._heapify_max(listForTree)     # Pour max_heap

listForTree
# [15, 11, 14, 9, 10, 13, 7, 8, 4, 2, 5, 12, 6, 3, 1]
```

TAS en Python (suite)

TAS : une autre possibilité dans Python :

- Sous Python, la classe **Queue** propose la sous-classe *PriorityQueue* qui implante un TAS.
- La classe Queue elle-même implante une file (FIFO) utilisable en particulier dans les Threads.
- Sous Python, **PriorityQueue** est implantée à base du module **heapq** mais propose également une interface (classique) **queue**.

→ Cette interface est en grande partie la différence majeure entre les deux :

D'une part, une conception Objet (et sa notation pour *PriorityQueue*) vs. le passage d'une liste en paramètre (que les opérateurs de *heapq* peuvent modifier).

→ Autrement dit, les fonctions de *heapq* demandent un paramètre vs. une utilisation par la notation objet pour *PriorityQueue*

TAS en Python (suite)

Exemples (avec queue et sa sous classe *PriorityQueue*)

```
import queue as Q

q = Q.PriorityQueue()
q.put(10)
q.put(1)
q.put(5)
while not q.empty():
    print(q.get())

# 1 5 10
```

☞ Remarquer l'ordre des insertions et celui d'affichage.

- Si on veut ajouter des tuples :

```
import queue as Q

q = Q.PriorityQueue()
q.put((10, 'ten'))
q.put((1, 'one'))
q.put((5, 'five'))
while not q.empty() : print(q.get())

# (1, 'one') (5, 'five') (10, 'ten')
```

→ La comparaison se fait sur les premiers éléments des couples.

En savoir plus : Transformation de la récursivité

Rappels et compléments sur la récursivité (induction)

- Avantages

- Analyse naturelle et intuitive de multiples problèmes
- Apport de la preuve des algorithmes facilité
- Quasi obligation d'utilisation dans certains problèmes (en particulier traitant des structures récursives comme les arbres, les graphes...)
- Similarité avec l'Induction

- Inconvénients

- Plus coûteux en espace mémoire et en temps de calcul si mal écrit.
- Moins accessible : demande un effort d'exercices

En savoir plus : Transformation de la récursivité (suite)

- Solution aux inconvénients :

Il existe des techniques fiables de transformation des schémas récursifs en schémas itératifs.

→ Ces techniques sont prouvées justes et complètes.

- La démarche à suivre :

- écrire un algorithme récursif
- le tester, valider, prouver (dans les applications critiques)
- lui appliquer les techniques de transformation pour obtenir une version itérative

☞ Un soucis : ces techniques ne sont pas à 100% automatisables (pour l'instant !)

→ Dans des cas non triviaux, une intervention humaine peut être nécessaire.

Récursivité terminale

- Il existe un lien direct entre un schéma itératif et celle récursive terminale.

Exemple : schéma de la récursivité terminale :

```
f_rec(X) =
    si p(X) alors a(X)
    sinon
        b(X)
        f_rec(nouveau(X))
    Finsi;
```

- Un exemple utilisant ce schéma : affichage d'une chaîne caractère par caractère

```
afficher(Ch) =
    si Ch="" alors RIEN
    sinon
        écrire(tête(Ch))
        afficher(Reste(Ch))
```

- La traduction du schéma récursif terminal sera de la forme :

```
f_iter(X) =
    Tant que p(X) = faux faire
        b(X);
        X := nouveau(X);
    Fin Tq;
    a(X);
```

Récursivité terminale (suite)

Remarque :

- Les schémas *Tant que* et *répéter* sont équivalents :

→ On propose un schéma répéter équivalent :

```
f_iter(X) =  
    si p(X) = faux alors  
        répéter  
            b(X);  
            X := nouveau(X);  
            jusqu'à p(X)=vrai;  
        a(X);
```

- On en déduit :

L'évaluation d'une fonction récursive terminale est une itération.

Récursivité terminale (suite)

- Remarque : un schéma récursif terminal est de la forme :

```
P(X) =  
    si cond(X) alors  
        action_A(X)  
        P(nouveau(X))  
    sinon  
        action_B(X)  
    Finsi;
```

- Cet algorithme peut être résumé par l'expression de la logique des prédictats :

$$P(x) : [Cond(x) \wedge Action_A(x) \wedge P(\rho(x))] \vee [\neg Cond(x) \wedge Action_B(x)]$$

Où la fonction $\rho(x)$ (notée *nouveau(x)*) modifie la valeur de x pour la faire converger vers $Cond(x) = vrai$

- De cette expression, on peut tirer l'arbre de développement suivant :

Récursivité terminale (suite)

- La séquence d'actions effectuées :

*cond(x₀), A(x₀), cond(x₁), A(x₁), cond(x₂), A(x₂),
..., cond(x_n), A(x_n), cond(x_{n+1}), B(x_{n+1})*

Où x_{i+1} est obtenue par $\rho(x_i)$

- Dont on peut déduire le schéma :

$i \leftarrow 0$

Tant que cond(x_i)

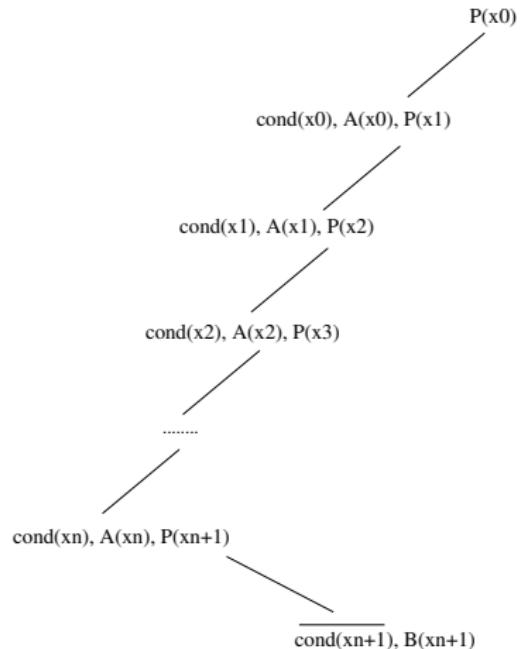
$A(x_i)$

$x_{i+1} \leftarrow \rho(x_i)$

Fin TQ

// ici : on a : cond(x_{n+1})

$B(x_{n+1})$



Récursivité terminale (suite)

Avec Python :

```
def p1_rec (i) :
    if i > 0 :
        action1(i)                      # ne contenant pas d'appel à p1_rec
        p1_rec(i-1)                      # (i non modifié par action1)
    else :
        action2(i)                      # ne contenant pas d'appel à p1_rec
```

- La solution itérative supprime la récursivité terminale.

```
def p1_iter (i) :
    j=i
    while j > 0 :
        action1(j)
        j = j-1
    action2(j);
```

Récursivité non terminale

Cas de récursivité non terminale :

```
def p2_rec (i) :
    if i > 0 :
        p2_rec(i-1)
        action1(i)          # ne contenant pas d'appel à p2_rec
    else :
        action2(i)          # ne contenant pas d'appel à p2_rec
```

- La version itérative utilise une **pile** (comme une *pile d'assiettes*) avec une mode de gestion :
le dernier arrivé est le premier servi.
- Les opérateurs habituellement définis sur les piles sont :
 - *vide* : la constante pile vide
 - *empiler(élément, pile)* : procédure d'ajout d'un élément au sommet ,
 - *dépiler(pile)* : procédure d'extraction du sommet (fonction partielle ?)
 - *sommet(pile)* : fonction d'interrogation de l'élément du sommet (partielle ?),
 - ...

Récursivité non terminale (suite)

Rappel du schéma récursif :

```
def p2_rec (i) :
    if i > 0 :
        p2_rec(i-1)
        action1(i)          # ne contenant pas d'appel à p2_rec
    else :
        action2(i)          # ne contenant pas d'appel à p2_rec
```

- Et le schéma itératif équivalent :

```
def p2_iter (i) :
    j=i;
    pile = []
    while i > 0 :
        empiler(j, pile)
        j=j-1

    while pile != vide :
        j=sommet(pile)
        depiler(pile)
        action1(j)
        action2(j)
```

Récursivité non terminale (suite)

- Remarque : pour un schéma récursif non terminal de la forme :

```
P(X) =
    si cond(X) alors
        P(nouveau(X))
        action_A(X)
    sinon
        action_B(X)
    Finsi;
```

L'expression associée en logique des prédicats sera (cf. la récursivité terminale) :

$$P(x) : [Cond(x) \wedge P(\rho(x)) \wedge Action_A(x)] \vee [\neg Cond(x) \wedge Action_B(x)]$$

Où la fonction $\rho(x)$ (notée *nouveau*(x)) modifie la valeur de x pour la faire converger vers $Cond(x) = \text{vrai}$

- On constate clairement que $Action_A(x)$ a besoin de l'ancienne valeur de x modifiée par $\rho(x)$. Par exemple, si $\rho(x) : x = x + 1$, il suffira de restaurer l'ancienne valeur de x en faisant $x - 1$ mais dans le cas général, on doit stocker les différentes valeurs de x dans une pile de sorte que l'on puisse les récupérer et leur appliquer $Action_A$.
- Une étude de l'arbre de développement de ce schéma montre que :

Récursivité non terminale (suite)

- La séquence d'actions effectuées :

cond(x₀), cond(x₁), cond(x₂), ..., cond(x_n)

cond(x_{n+1}), B(x_{n+1}) \leftarrow la 1^e action est *B(x_{n+1})*

A(x_n), A(x_{n-1}), ..., A(x₂), A(x₁), A(x₀)

Où x_{i+1} est obtenue par $\rho(x_i)$

- Dont on peut déduire le schéma :

$i \leftarrow 0; Pile \leftarrow \text{vide}$

Tant que *cond(x_i)*

empiler(x_i, Pile)

$x_{i+1} \leftarrow \rho(x_i)$

Fin TQ

// ici : on a : *cond(x_{n+1})*

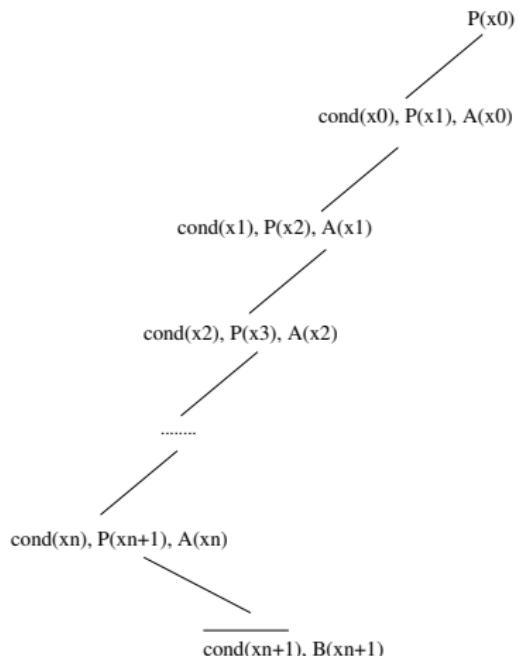
B(x_{n+1}) \leftarrow la 1^e action

Tant que *est_vide(Pile) = faux*

$x_i \leftarrow \text{sommet}(Pile); \text{depiler}(Pile)$

A(x_i) \leftarrow la 1^e valeur est *A(x_n)*

Fin TQ



Récursivité non terminale : exemples

Exemple : conversion décimale-binaire

- On souhaite afficher la séquence de bits (0 ou 1) qui représente la forme binaire d'un entier positif X.
- Par exemple, l'entier 5 donnera lieu à la séquence 101.
- Le code ci-dessous est équivalent aux divisions successives que l'on ferait pour atteindre manuellement le même objectif.

```
def affiche_bin_rec(X)
    if X==1 : print('1')
    else :
        affiche_bin_rec(X//2)
        if (X %2 == 0) : print('0')
        else : print('1')
```

Récursivité non terminale : exemples (suite)

- Une trace pour X=17 donnera la chaîne de bits 10001.
- La trace des appels ressemble à :

```
P(17):
 P(8):
 P(4):
 P(2):
 P(1):
      1
     0
    0   0
   1
```

- Résultat pour $17 = 10001$
- Selon le schéma général de la transformation d'une récursivité non terminale, la version itérative de cette fonction sera (on utilise une liste Python pour représenter la Pile) :

```
def affiche_bin_iter(X) :
    Pile=[]
    while (X > 1) :
        Pile.append(X)
        X=X//2
    if (X==1) : print("1")
    while (Pile != []) :
        X=Pile.pop()           # cette fonction fait sommet + dépile !
        if (X %2 == 0) : print("0")
        else : print("1")
```

Récursivité non terminale : exemples (suite)

Exemple factorielle :

```
def FACT(N) :
    if (N>1) :
        temp= FACT(N-1)
        Val= temp * N
    else :
        Val =1
    return Val;
```

- La multiplication ne peut avoir lieu que lorsque la valeur Fact(N-1) est disponible, ce qui fait de cet appel un cas non terminal.
- Le schéma itératif :

```
def FACT_iter (N) :
    Pile=[]
    while (N>1) :
        Pile.append(N)
        N=N-1

    val=1
    while (Pile != []) :
        N = Pile.pop()      # cette fonction fait sommet + dépile !
        val= val * N

    return val;
```

Récursivité non terminale : exemples (suite)

Cas particulier d'énumération de valeurs

- Ci-dessus, on a indiqué que dans certains cas numériques, on peut éviter la Pile.
- Exemple de transformation de la fonction factorielle sans utiliser une pile. Dans certains cas, on peut éviter d'utiliser une pile si son utilité est simplement de contenir différentes valeurs successives (ou régulier, p.e. les nombres pairs) d'un entier.
- Rappel de l'exemple factorielle où l'on constate bien la place de l'appel récursif :

```
def FACTORIELLE (N) :  
    if (N>1) :  
        temp= FACTORIELLE (N-1)  
        Val= N * temp  
    else :  
        Val =1  
    return Val
```

- La multiplication ne peut avoir lieu que lorsque la valeur *FACTORIELLE(N-1)* est disponible, ce qui fait de cet appel un cas non terminal.

Récursivité non terminale : exemples (suite)

- La version itérative obtenue par une transformation qui utilise une Pile.

```
def FACTORIELLE_iter(N) :
    Pile = []
    while (N>1) :
        Pile.append(N)
        N=N-1
    val=1
    while (Pile != []) :
        N=Pile.pop()
        val= val * N

    return val;
}
```

- Du fait de faire $N-1$ fois ' -1 ' sur N puis autant de fois retirer ces valeurs de la Pile avant de les multiplier, on peut proposer une solution qui n'utilise pas une pile :

```
def FACTORIELLE_iter_sans_pile(N) :
    while (N>1) :
        N=N-1
    compteur = compteur +1 # compte (init 0) est le nbr de fois où on a fait l'action "N=N-1"

    val =1;                      # N vaut 1
    while (compteur > 0) :
        N=N+1
        val= val * N
        compteur = compteur -1
    return val;
```

Exemples non triviaux

Exemple du schéma Hanoï :

- L'exemple des tours de Hanoï est doublement intéressant car il contient à la fois une récursivité terminale et une non terminale.

```
def HANOI(N, dep, arr, inter) :
    if (n > 0) :
        HANOI(n-1 , dep, inter , arr)
        print( dep, "→", arr)
        HANOI(n-1 , inter , arr, dep)
```

- On procède en 2 étapes :
 - élimination de la récursivité terminale puis celle de la récursivité non terminale.

Hanoï : élimination de la récursivité terminale

- En suivant les indications ci-dessus, on obtient la version hanoi_iter1 :

```
def hanoi_iter1(N, dep, arr, inter) :
    while (N>0) :
        hanoi_iter1(N-1, dep, inter , arr)
        print( dep, "→", arr)
        N=N-1
        permuter(dep, inter)
```

Exemples non triviaux (suite)

- On constate qu'il est nécessaire de permutez *dep* et *inter* du fait de l'appel récursif terminal *hanoi_rec(N-1, inter, arr, dep)*.
- Noter que l'appel récursif non terminal s'inscrit dans une boucle `while`.
 - Dans les cas similaires, on a recours à la suppression du *while* (ou *for*, etc.)

Hanoï : élimination de la récursivité non terminale

- Pour simplifier cette transformation, considérons la version suivante de *hanoi_iter1* où la boucle `while` a été remplacée par un *if + goto*.

```
void hanoi_iter1_avec_if_et_label(N, dep, arr, inter) :
label:
  if (N>0) :
    hanoi_iter1_avec_if_et_label(N-1, dep, inter, arr)
    print( dep, "→", arr)
    N=N-1
    permuter(dep, inter)
    goto label
```

- De cette manière, on peut appliquer le schéma de transformation vu ci-dessus.
- En utilisant une pile, on obtient (supposons disposer des labels en Python) :

Exemples non triviaux (suite)

```

void hanoi_iter2_avec_label(N, dep, arr, inter) :
    Pile = []
    label : while (N>0) :
        Pile.append([N, dep, arr, inter])
        N=N-1
        permuter(arr,inter)      # du fait de l'appel récursif
    }
    if(Pile != []):
        data=Pile.pop()
        print( dep, "—>", arr)   # devenu "if" à cause du goto label ci-dessous
        # Mêmes remarques que ci-dessus
        N=data[0]-1
        permuter(dep,inter)
        goto label
# équivalent à N = N - 1

```

- On peut maintenant donner la version qui nous débarrasse du label (remplacé par *while*) :

```

void hanoi_iter2(N, dep, arr, inter) :
    Pile = []
    while True :
        while (N>0) :
            Pile.append([N, dep, arr, inter])
            N=N-1
            permuter(arr,inter)      #du fait de l'appel récursif

        if(Pile != []):
            data=Pile.pop()
            print( dep, "—>", arr)   # équivalent à N = N - 1
            data[0]-1
            permuter(dep,inter)

        if (N <= 0 or Pile==[]) : break

```

Exemples non triviaux (suite)

Exemple des primes :

- On rappelle l'exemple de distribution des primes pour une transformation.
- On veut distribuer une prime de N francs entre M les employés d'une entreprise.
- Soit $e_1 \dots e_m$ la série ordonnée qui représente le personnel dans l'ordre de leur ancienneté dont la distribution doit tenir compte : le montant affecté à e_i doit être $\geq e_{i+1}$.
- Certaines primes peuvent être nulles.
- Déterminer le montant affecté à chaque personne ainsi que le nombre de manières de répartir N en M sommants. .

Rappel de la solution récursive :

- La solution récursive est rappelée en séparant les différentes sections du code Python :
 - Les tests ont été séparés et placés dans une fonction `cond()` pour plus de clarté.

Exemples non triviaux (suite)

```

def cond(Montant_restant, Empl_restants, Last_montant) :
    if (Empl_restants < 1) or (Montant_restant < 0) : return False
    elif Empl_restants == 1 :
        if Montant_restant > Last_montant : return False
        else : return True # Montant_restant <= Last_montant
    elif Montant_restant == 0 : return True # Empl_restants > 1
    return True

def prime3(Montant_restant, Empl_restants, Last_montant) :
    Res=0
    if cond(Montant_restant, Empl_restants, Last_montant) :
        # »» Action A(x) ««
        Min_ = min(Last_montant, Montant_restant) # min de 2 valeurs

        N = 0 # Pour la boucle, ne fait pas partie de A(x), noter les "vraies" actions
        # Partie récursive non terminale
        for K in reversed(range(1, Min_+1)) :
            # K donné à l'employé actuel, on va distribuer le reste
            temp = prime3(Montant_restant-K, Empl_restants-1, K)

            # »» Action B(x) ««
            N += temp
        Res=N
    else :
        # »» Action D(x) ««
        if (Empl_restants < 1) or (Montant_restant < 0) : Res=0
        elif Empl_restants == 1 :
            if Montant_restant > Last_montant : Res=0
            else : Res=1 # Montant_restant <= Last_montant
        elif Montant_restant == 0 : Res=1 # Empl_restants > 1
    # »» Action E(x) ««
    return Res

```

Exemples non triviaux (suite)

- La schéma récursif généralisé est alors de la forme :

```
def P_rec_non_terminale(X) :
    if cond(X) :
        Action A(x)
        for K in range(Inf..Sup) :
            P_rec_non_terminale(nouveau(X))      # le vecteur X modifié
            Action B(x)

    else :
        Action D(x)

    Action E(x)
```

Exemples non triviaux (suite)

- Le schéma itératif correspondant sera (voir aussi le schéma Hanoï) :

```

def P_iteratif(X) :
    Pile=[]
    while True :
        if cond(X) :           # Condition pour empiler des valeurs
            Action A(x)

            for K in range(Inf..Sup) :
                # Empiler différentes configurations issues du vecteur X
                # On les traite ensuite (noter nouveaux(X) empilé)
                empiler(Pile , nouveau(X))

            Action D(x)

        Pile_vide = (Pile == [])
        if (Pile != []) :      # Simple précaution avant de dépiler
            X = sommet(Pile)
            depiler(Pile)

        Action B(x)
        if Pile_vide : break   # Un test (Pile != []) ne suffit pas car on
                               # vient peut être de dépiler et vidé la Pile

    Action E(x)

```

Exemples non triviaux (suite)

```

def prime_iter(Montant_restant, Empl_restants, Last_montant) :
    Res = 0
    Pile = []
    while True :
        if cond(Montant_restant, Empl_restants, Last_montant) :

            # Action A(x)
            Min_ = min(Last_montant, Montant_restant)

            # Appel Récursif non terminale transformée
            for k in reversed(range(1, Min_+1)) :
                Pile.append([Montant_restant-k, Empl_restants-1, k])

            # Action D(x)
            N=0
            if (Empl_restants < 1) or (Montant_restant < 0) : N = 0
            elif Empl_restants == 1 :
                if Montant_restant > Last_montant : N = 0
                else : N = 1                      # Montant_restant <= Last_montant
            elif Montant_restant == 0 : N = 1      # Empl_restants > 1

            Pile_vide= (Pile == [])
            if (Pile != []) :
                [Montant_restant, Empl_restants, Last_montant] = Pile.pop()

            # Action B(x)
            Res += N
            if Pile_vide : break

        # Action E(x)
        return Res

print(prime_iter(10,5,10)) # 30 OK

```

Exemples non triviaux (suite)

Remarques :

- Il n'y a pas de transformation automatisée à 100%, sauf dans les cas simples.
- Sachant qu'il y a une grande diversité d'algorithmes récursifs (il y a autant de versions que d'algorithmes différents), il faut adapter le schéma itératif aux particularités du schéma récursif en main.

Addendum : Parcours itératif d'arbre binaire

S'obtient par la transformation de schéma récursif.

Exemple : traduction itérative du parcours récursive infixé.

```
Procédure Infixe ( R : Arbre_bin ) =
Début
    Si non est_vide(R) alors
        Infixe (gauche(R));
        traiter(info(R));
        Infixe (droit(R));
    Fin si;
Fin Infixe ;
```

- La première étape de transformation supprime la récursivité terminale :

```
Procédure infixe1(Racine : Arbre_bin) =
R ← Racine : Arbre_bin;
Début
    Tant que non est_vide(R)
        infixe1 (gauche(R))
        traiter (info(R));
        R ← droit(R);
    Fin Tq;
Fin infixe1;
```

Addendum : Parcours itératif d'arbre binaire (suite)

- La seconde étape supprime la récursivité non terminale à l'aide d'une Pile.
- On utilisera les opérateurs habituels de la Pile :

```
Procédure infixe2(Racine : Arbre_bin) =  
R <- Racine : Arbre_bin;  
P <- Pile_vide : Pile;  
Début  
    Tant que non est_vide(R) ET non Est_vide(P)  
        Tant que non est_vide(R)  
            Empiler(P, R);  
            R <- gauche(R);  
        Fin Tq;  
        Dépiler(P, R);  
        traiter(info(R));  
        R <- droit(R);  
    Fin Tq;  
Fin infixe2
```

Exceptions : Traitement des erreurs, ruptures

- Un programme ne se déroule pas toujours comme prévu
- Certaines fonctions (dites partielles) ne s'appliquent que sous condition.
- Plusieurs cas d'erreurs perturbateurs :
 - Division par zéro,
 - Dépassement des bornes d'un tableau/liste,
 - Débordement de la pile,
 - Fichier inexistant,
 - Erreur de type repérée à l'exécution...
- Vous en avez déjà vu :
 - Accès à un élément d'une liste dépassant sa capacité
 - Division par zéro
 - Lecture de caractères alphabétiques là où il faut des chiffres
 - Dépassement de la limite de la récursivité Python
 - Fichier à ouvrir absent
 - Etc

Exceptions : Traitement des erreurs, ruptures (suite)

- Traitements classiques :
 - Appel d'une procédure d'erreur
 - Les difficultés pour rester dans le bloc en cours
 - Hiérarchisation
 - Qui traite l'erreur : l'appelant ou l'appelé
- ☞ Parfois, en cas d'erreur, on ne peut pas provoquer un arrêt du programme :
 - On peut arrêter un ascenseur / escalateur en cas de bug
 - Mais dans un système embarqué sensible, l'arrêt peut être grave
 - Qu'est-ce que d'arrêter une fusée, un avion, une chaîne de production
- Il faut donc être capable de tolérer les erreurs et pannes, prévoir leur traitement.
- ☞ Dans de multiples cas, Python **impose** de prévoir le traitement d'une exception possible.

Exceptions : Traitement des erreurs, ruptures (suite)

Et en Python : Exception

- Un outil pour résoudre ce genre de problèmes.
- Déclenchement d'une exception par :
 - le matériel (division par zéro) ;
 - l'exécutif JVM (erreur de fichier)
- les exceptions prédéfinies qui se déclenchent automatiquement
 - le programme lui même (nous quoi !)
- les exception définies dans le programme déclenchées explicitement (par RAISE).

Exceptions : Traitement des erreurs, ruptures (suite)

Sous Python :

Ces déclenchements peuvent avoir lieu pour signaler un cas inhabituel (exceptionnel).

- Notion de traiteur (récupérateur) d'exceptions
- Propagation hiérarchique
- Re-propagation vers l'appelant (traitement + Raise)

Différents variétés d'exceptions

- prédefinie
- fournie par un module (prédefini ou non)
- déclarée par l'utilisateur

☞ Rappel : les erreurs sémantiques échappent aux exceptions :

→ si vous décidez que $2 + 2 = 5$, Python ne vous dira rien !

Exceptions Python

Les exceptions prédéfinies de Python (extrait de la doc) :

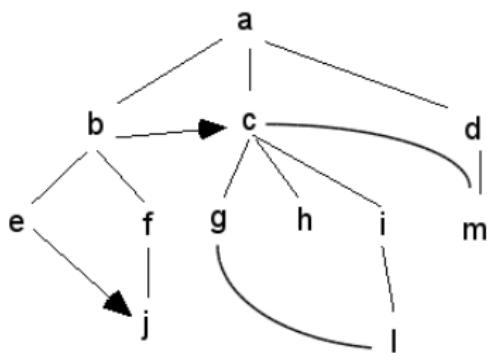
```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- ArithmeticError
        /    +-- FloatingPointError
        /    +-- OverflowError
        /    +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EnvironmentError
        /    +-- IOError
        /    +-- OSError
        /    +-- WindowsError (Windows)
        /    +-- VMSError (VMS)
    +-- EOFError
    +-- ImportError
    +-- LookupError
        /    +-- IndexError
        /    +-- KeyError
    +-- MemoryError
    +-- NameError
        /    +-- UnboundLocalError
    +-- ReferenceError
    +-- RuntimeError
        /    +-- NotImplementedError
    +-- SyntaxError
```

Exceptions Python (suite)

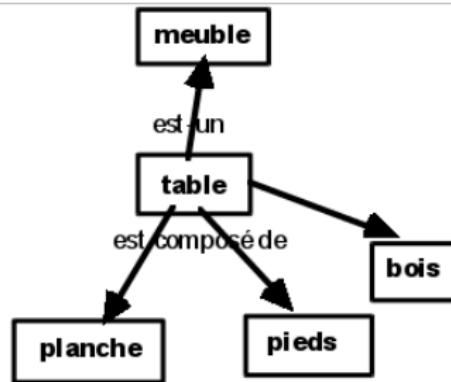
```
/   +-- IndentationError
/   +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
/   +-- UnicodeError
/   +-- UnicodeDecodeError
/   +-- UnicodeEncodeError
/   +-- UnicodeTranslateError
+
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
```

Graphes

- Exemples :



un graphe représentant des liens entre différents noeuds (villes)



un graphe représentant des liens d'héritage et de composition

Graphes (suite)

Quelques exemples d'applications :

- Recherche de chemin : Dijkstra, Floyd Warshall, etc
 - Table de routage : chemins entre les routeurs, recherche de meilleures voies.
- Efficacité des pipelines : FLUX, MST
- Messagerie : le voyageurs de commerce , trajet d'un facteur
- Réseaux de communication : MST
- Gestion du trafic : problème de FLUX, chemins d'encombrement minimum
- Navigation aérienne (les avions dans des couloirs au ciel !)
- Le système de transport fermé (circuit fermé) : livraison de marchandises, TSP.
- Câblage de circuits imprimés
- etc...

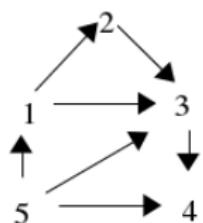
Graphes (suite)

Quelques définitions

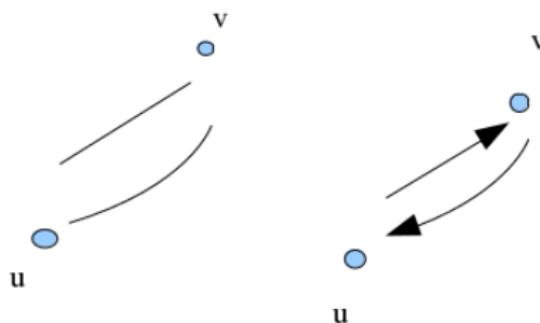
- Un graphe $G=(V, E)$

V : ensemble de noeuds (vertex)

$E \in (V \times V)$: ensemble d'arêtes ou arcs (edges)



Un Digraphe

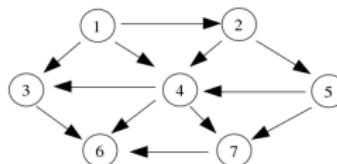


Graphes (suite)

- Chaque **arc** = une paire $(v, w) \in E, v, w \in V$
 - Si (v, w) est **ordonné**, on aura un graphe **orienté** (*digraphe*)
 - *directed graph*
 - w est **adjacent** de v si $(v, w) \in E$
 - Dans un graphe non orienté (undirected), $(v,w) \sim (w,v)$ v et w sont adjacents
 - Dans certains problèmes, les noeuds représentent les *variables* et les arcs les *relations*.
 - **Poids** (*weight*, pondération) : valeur d'un arc/arête
 - **Chemin** (branche, path) : w_1, w_2, \dots, w_n tel que $(w_i, w_{i+1}) \in E$
 $\text{chemin}((X_1, \dots, X_k), (V, E)) \equiv (X_1, X_2) \in E \wedge \dots \wedge (X_{k-1}, X_k) \in E$
 - **Longueur** d'un chemin contenant N noeuds=nombre d'arcs=n-1
 - Un noeud Y est **accessible** depuis un noeud X s'il existe un chemin de X vers Y :
 $(X, Y) \in E \vee \exists Z_1, \dots, Z_k : \text{chemin}((X, Z_1, \dots, Z_k, Y), (V, E))$
 - Un **arbre** : un graphe acyclique connexe
 - Un graphe est **dense** si $|E| = O(|V|^2)$
- voir les annexes pour un complément

Graphes (suite)

- Exemple de graphe



Graphe G1 : graphe orienté du trafic

Représentation d'un graphe :

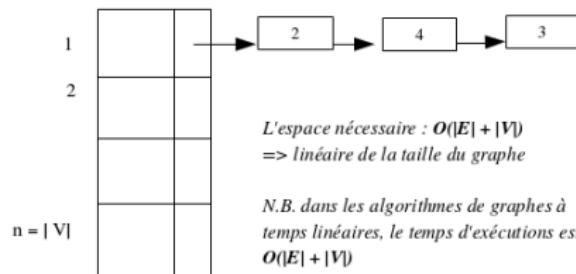
- Par une matrice carrée
 - Par un tableau/liste principal + tableau/liste des adjacents
 - Par un Dictionnaire
 - etc.
- Selon la nature du graphe (information contenue, orienté ou non, valué ou non), les structures peuvent être plus ou moins complexes.

Représentation abstraite des graphes

- Représentation par une matrice (ou liste de listes)

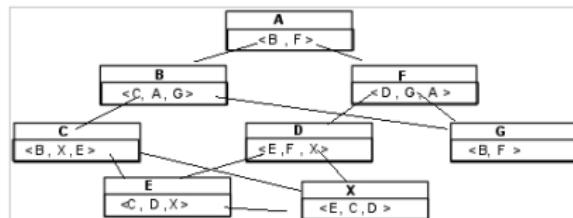
| | A | B | C | D | E | F | G |
|---|---|----|---|---|----|---|---|
| A | | | | | | | |
| B | | | V | | | | |
| C | | F | | | | | |
| D | | 15 | | | 3 | | |
| E | | | | | -1 | | |
| F | | | | | | | |
| G | | | | | | | |

- Représentation par un dictionnaire (avec listes d'adjacents)



Représentation abstraite des graphes (suite)

- Un exemple de graphe ...



- .. Et sa représentation par un dictionnaire ou liste de listes :

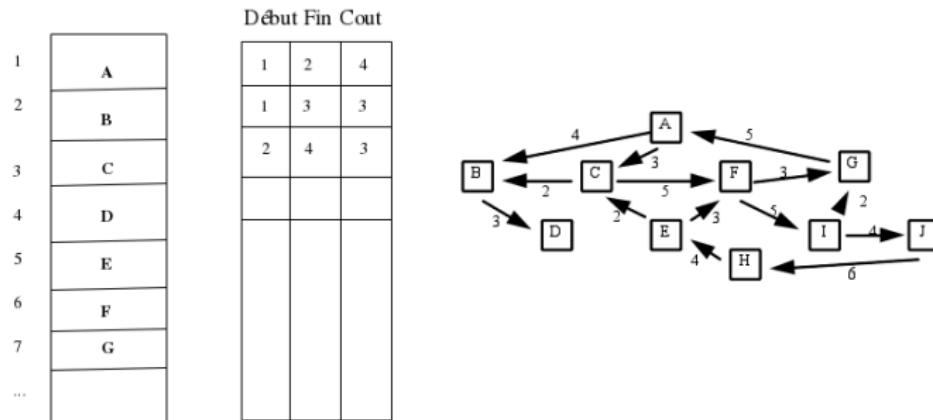
→ La colonne 1 contient toute information

| | noeud et ses données | liste d'indices des adjacents |
|---|----------------------------------|-------------------------------|
| 1 | Nom et données du noeud numéro 1 | <i, k,> |
| i | | |
| k | | |
| n | | |

- ☞ De manière similaire, au lieu de dupliquer les informations des noeuds, on utilise la synonymie (de Python) : ne pas copier les listes en profondeur.

Représentation abstraite des graphes (suite)

- Une représentation de graphe $G=(V, E)$ par un tableau / liste principale (contient les informations) et une matrice d'adjacents + couts.



La table gauche représente l'ensemble des noeuds V et la table droite représente l'ensemble E .

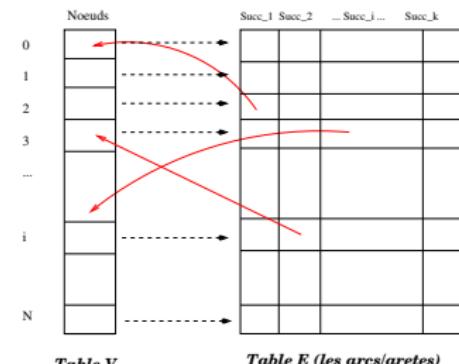
Dans chacune des lignes de la table E , on a l'arc $e_i \in E = (V_{debut} \times V_{fin})$ suivi du cout de cet arc.

Représentation abstraite des graphes (suite)

Plus concrètement :

- Le graphe $G(V, E)$ (avec V : ensemble des noeuds, E : ensemble d'arcs/arêtes) est représenté par ces deux tables.
- Habituellement, la table V contient toutes les informations sur les noeuds (p. ex. une ville, sa population, sa superficie, ...).
- La table E n'a pas besoin de répéter ces informations et se contente de contenir le nom du successeur, ou mieux, l'indice du successeur dans la table V .

- Les flèches **rouges** dans la figure ci-contre renvoient vers le noeud successeur dans V : ce renvoi se fait via le nom du *successeur_i* (le même nom que dans V) ou via l'indice du successeur dans V .
- Si une pondération des arcs (arêtes) est présente, chaque case de E sera un couple (*noeud_succ, poids*).



Parcours général de graphes

- Deux types majeurs de parcours :
 - 1- En profondeur : avantages en inconvénients
 - 2- En largeur : avantages et inconvénients

Il y a également des parcours ad-hoc (B&B, A*, etc)
- Des deux types de parcours notables, on peut obtenir des parcours variés (en particulier dans le cadre d'un parcours en largeur).
- **Remarque** : comme pour les arbres et selon le "moment" où l'on traite l'information d'un noeud, on a différents types de traitements : pré-ordre, post-ordre et mi-ordre.

Le principe de parcours récursif en profondeur (pré-ordre)

- Traiter chaque noeud puis traiter son premier adjacent récursivement avant de traiter les autres adjacents,
- Éviter de retraiter un noeud en marquant les noeuds visités

Parcours général de graphes (suite)

Parcours en profondeur :

- Ce parcours est semblable au parcours en profondeur des arbres.
- Il est en général assez performant mais si le graphe contient un cycle "à gauche", alors aucune réponse ne pourra être produite.
- Pseudo code du parcours en profondeur (version de base, **sans marquage**) :

```

Procédure profondeur ( G : ref Noeud ) =
Début
    Si Non est_vide(G) alors
        traiter(noeud_courant(G));           // un traitement quelconque
        Pour X dans adjacents(noeud_courant(G))
            profondeur (X);
        Fin pour;
    Fin si;
Fin profondeur ;

```

- Ce parcours ne mémorise rien et peut donc entrer dans une boucle infinie en cas de circuit ou une boucle (loop).

Parcours général de graphes (suite)

- Parcours récursif en profondeur **avec marquage**

```
Procédure profondeur (G : ref Noeud) =  
Début  
    Si Non est_vide(G) alors  
        marquer(noeud_courant(G));           # on marque tout de suite et ...  
        traiter(noeud_courant(G));          #traitement quelconque  
        Pour X dans adjacents(noeud_courant(G))  
            Si X n'est pas marqué           # ... on contrôle ici  
                Alors profondeur(X);  
            Fin si;  
            Fin pour;  
        Fin si;  
    Fin profondeur ;
```

- Ce parcours marque les noeuds déjà visités pour ne pas les réessayer.
- Les mécanismes de marquage :
 - marquage à l'intérieur du noeud
 - marquage par une structure de données externe au graphe (un tableau, ...)

Parcours général de graphes (suite)

- Une autre manière de parcourir le graphe en marquant les noeuds est :

```

Procédure profondeur (G : ref Noeud) =
Début
    Si Non est_vide (G) ET noeud_courant(G) n'est pas marqué      # On contrôle en entrant
    Alors
        marquer(noeud_courant(G));          # et on marque ici
        traiter(noeud_courant(G));          # traitement quelconque
        Pour X dans adjacents(noeud_courant(G))
            profondeur(X);
        Fin pour;
    Fin si;
Fin profondeur ;

```

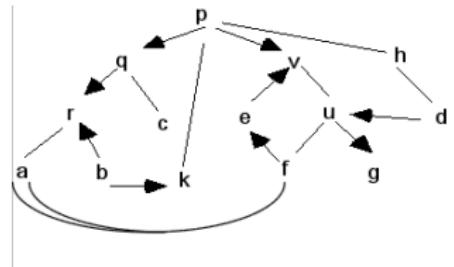
- Trace de parcours en profondeur de **p** à **u** :

$\text{profondeur}(p) \rightarrow \text{profondeur}(q) \rightarrow$

$\text{profondeur}(r) \rightarrow \text{profondeur}(a) \rightarrow$

$\text{profondeur}(f) \rightarrow \text{profondeur}(e) \rightarrow$

$\text{profondeur}(v) \rightarrow \text{profondeur}(u)$



Parcours général de graphes (suite)

- **Principe de l'algorithme itératif en Profondeur :**

- On a besoin d'une pile (LIFO) pour simuler la pile machine (expliquer).

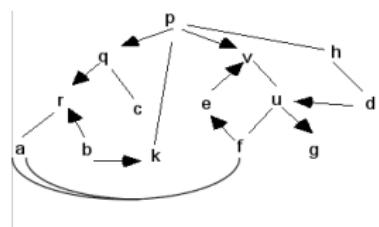
```

Procédure profondeur_iteratif(G)
    Pile = vide
    empiler(noeud_courant(G))
    Tant que NON est_vide(Pile)
        Noeud <- dépiler(Pile)
        Si NON est_marqué(X)           # Utile si un noeud se trouve 2 fois dans la Pile
            marquer(Noeud)
            traiter(Noeud)
            Pour X dans adjacents(Noeud)   # A considérer dans l'ordre voulu (p.ex. inversé = de gche à dte, V. Trace)
                Si NON est_marqué(X)
                    Alors empiler(Pile, X)      # empiler dans le désordre
                fin pour
        Fin Tant que
    Fin profondeur_iteratif

```

- Trace de parcours en profondeur de **p** à **d** (on souligne si traité) :

$[p] \rightarrow [h, v, k, \underline{q}] \rightarrow [h, v, k, c, \underline{r}] \rightarrow [h, v, k, c, \underline{r}] \rightarrow [h, v, k, c, \underline{a}] \rightarrow [h, v, k, c, \underline{f}]$
 $\rightarrow [h, v, k, c, e, \underline{u}] \rightarrow [h, v, k, c, e, \underline{g}, \underline{v}] \quad \rightarrow v \text{ se trouve 2 fois dans la pile sans être marqué}$
 $\rightarrow [h, v, k, c, e, \underline{g}] \rightarrow [h, v, k, c, \underline{e}] \rightarrow [h, v, k, \underline{c}] \rightarrow [h, v, \underline{k}] \rightarrow [h, \underline{v}]$
 $\quad \rightarrow v : \text{la 2e fois ! déjà traité.} \quad \text{puis}$
 $\rightarrow [\underline{h}] \rightarrow [\underline{d}] \rightarrow []$



Parcours général de graphes (suite)

Le principe de parcours en largeur

- traiter par niveau : traiter chaque noeud
- puis traiter chacun de ses successeurs avant de traiter les successeurs du prochain niveau.

☞ **Par contre :** contrairement au parcours en profondeur,
s'il existe un cycle dans le graphe, des réponses seront néanmoins produites.

- Ce parcours s'adapte mieux à une solution itérative.
→ On utilise une file d'attente pour la construction de la liste des noeuds à traiter.
- Les opérations sur une file :
 - enfiler(X, File)
 - defiler(File)
 - premier(File)
 - ...

Parcours général de graphes (suite)

Pseudo algorithme récursif de parcours en largeur (graphe connexe) :

- Remarque : certains noeuds sont traités plusieurs fois, voir marquage

```

File : file d'attente = File_vide;

Procédure largeur_récuratif (G : ref Noeud) =
Début
    Si Non est_vide(G) alors
        traiter(noeud_courant(G));           //une opération quelconque
        Pour X dans adjacents(noeud_courant(G))
            File=Enfiler(X, File);
        Fin pour;
        X = Sommet(File);                  // Respect du TDA File
        File = Défiler(File);
        largeur_récuratif (X);
    Fin si;
Fin largeur_récuratif ;

```

☞ N.B. : pas de marquage

→ en l'absence de marquage, certains noeuds sont traités plusieurs fois.

Parcours général de graphes (suite)

Cas de graphe connexe : tout noeud est connecté aux autres.

- on peut travailler (se laisser guider) directement par la file d'attente

```

File : file d'attente = File_vide;
enfiler(la racine du graphe G, File)

Procédure largeur_récuratif (G, File) =
Début
    Si Non est_vide(File) alors
        X=Sommel(File);
        File=Défiler(File);           // car Défiler ne renvoie pas un élément (cf. TDA File)
        traiter(noeud_courant(X));     //un traitement quelconque
        Pour X dans adjacents(noeud_courant(G))
            File=Enfiler(X, File);
        Fin pour;
        largeur_récuratif (G, File);
    Fin si;
Fin largeur_récuratif ;

```

- Initialement, il faut enfiler le noeud de départ.
- ☞ Le marquage (pour éviter de re-traiter un noeud) se fait comme pour le parcours en profondeur (voir ci-après).

Parcours général de graphes (suite)

- On peut récupérer le chemin par le mécanisme *Coming-From*.
- Pour un graphe connexe, la version suivante récupère le chemin par ce mécanisme.

```

File : file d'attente =File_vide ;
enfiler(la racine du graphe G, File)
CF : tableau indicé par les noeuds initialisé à 0
CF[Départ]=Départ

Procédure chemin_largeur_récuratif (File) =
Début
    Si Non est_vide(File) alors
        X=Sommet(File);
        File=Défiler(File);           // car défiler ne renvoie pas un élément (cf. TDA File)
        traiter(noeud_courant(X));    //une opération quelconque
        Pour X dans adjacents(noeud_courant(G))
            File=Enfiler(X, File);
            CF[X]=noeud_courant(G)
        Fin pour;
        largeur_récuratif (File);
    Fin si;
Fin largeur_récuratif ;

```

- Exercice : comment extraire ensuite le chemin ?
- Note : voir la version Python du problème de la monnaie.

Parcours général de graphes (suite)

L'algorithme itératif de parcours en largeur

```
File : file d'attente=File_vide;

Procédure largeur_itératif ( G : graphe)
    File=vide
    Début
        Si Non est_vide(G) alors
            Enfiler(noeud_courant(G) , File);
        Fin si;
        Tant que Non est_vide(File)
            N = Sommet(File);
            File = Défiler(File);
            traiter(N);      // ou visiter(N)
            Pour X dans adjacents(N)
                File = Enfiler(X, File);
            Fin pour;
        Fin Tant que;
    Fin largeur_itératif ;
```

☞ N.B. : pas de marquage

Parcours général de graphes (suite)

Trace de parcours en largeur

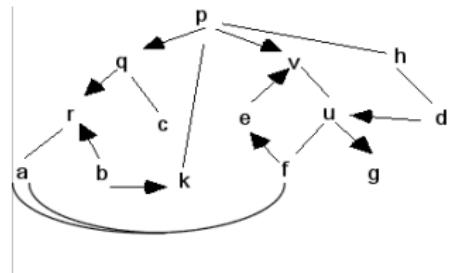
- Parcours en largeur de p à u

(noter le cycle $P \rightarrow \dots \rightarrow P \dots$) :

largeur(p) → largeur(q) → largeur(k) →

largeur(v) → largeur(h) → largeur(r) →

largeur(c) → largeur(p) → largeur(u)



- L'évolution de la File lors de ce parcours : ajout tant que $sommet \neq u$:

$[] \rightarrow [p] \rightarrow [q, k, v, h] \rightarrow [k, v, h, r, c] \rightarrow [v, h, r, c, p] \rightarrow [h, r, c, p, u]$

$\rightarrow [r, c, p, u, p, d] \rightarrow [c, p, u, p, d, a] \rightarrow [p, u, p, d, a] \rightarrow [u, p, d, a, q, k, v, h]$

→ Destination atteinte.

Parcours général de graphes (suite)

L'algorithme itératif de parcours en largeur avec marquage

Ici, la File peut contenir des doublons qui ne seront pas traités une seconde fois.

```

File : file d'attente=File_vide ;
Procédure largeur_itératif_marquage ( G : graphe )
Début
    Si Non est_vide(G)
        Alors
            Enfiler(noeud_courant(G), File) ;
        Fin si ;
        Tant que Non est_vide(File)
            N = Sommet(File) ;
            File = Défiler(File) ;
            Si est_marqué(N)
                Alors passer à l'itération suivante ;
            Sinon marquer(N) ;
            Fin si ;
            traiter(N); // ou visiter(N)
            Pour X dans adjacents(N):
                File=Enfiler(X, File) ;
            Fin pour;
        Fin Tant que;
    Fin largeur_itératif_marquage ;

```

Parcours général de graphes (suite)

Une variante parcours en largeur itératif (avec marquage)

- Une seconde version avec marquage :
 - on marque les noeuds non marqués placés dans la file.
- La file ne contiendra pas de doublon.

```

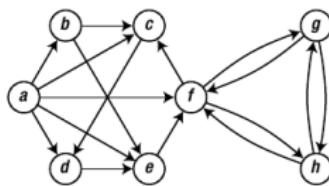
File : file d'attente=File_vide;
Procédure premier_chemin_largeur_itératif ( G : graphe)
Début
  Si Non est_vide(G) alors
    Enfiler(noeud_courant(G), File);
    marquer(noeud_courant(G));
  Fin si;
  Tant que Non est_vide(File)
    N = Sommet(File);
    File = Défiler(File);
    traiter(N); // ou visiter(N)
    Pour X dans adjacents(N) non marqués
      File = Enfiler(X, File);
      marquer(X);
    Fin pour;
  Fin Tant que;
Fin premier_chemin_largeur_itératif ;

```

Représentation par ensembles d'adjacents

☞ Pléthore de représentations possibles.

- Cas de graphe non valué (non pondéré) représenté par une liste d'ensembles.



```
a, b, c, d, e, f, g, h = range(8)
N = [
    {b, c, d, e, f}, # a
    {c, e}, # b
    {d}, # c
    {e}, # d
    {f}, # e
    {c, g, h}, # f
    {f, h}, # g
    {f, g} # h
]
```

Représentation par listes d'adjacents

- Une autre présentation des graphes comme liste (ou array) d'adjacents.
On dit array car les listes de Python sont des arrays dynamiques.

```
a, b, c, d, e, f, g, h = range(8)
N = [
    [b, c, d, e, f], # a
    [c, e], # b
    [d], # c
    [e], # d
    [f], # e
    [c, g, h],# f
    [f, h], # g
    [f, g] # h
]
b in N[a] # Neighborhood membership
# True
len(N[f]) # Degree
#3
```

- Une variation dans la représentation du graphe est de représenter les ensembles de voisins comme une liste triée. En particulier, si on ne modifie pas trop le graphe.

→ Permet une méthode dichotomique dans la recherche d'un voisin.

N.B. : les ensembles (set) prédéfinis de Python sont plus pratiques dans ce cas.

Représentation par dict : graphe valué

- Pour tenir compte des valeurs, utiliser les `dicts` de Python où le voisin sera la clé et la valeur permet de réaliser un graphe valué (avec arcs pondérés). Par exemple :

```
a, b, c, d, e, f, g, h = range(8)
N = [
    {b:2, c:1, d:3, e:9, f:4}, # a
    {c:4, e:3}, # b
    {d:8}, # c
    {e:7}, # d
    {f:5}, # e
    {c:2, g:2, h:2}, # f
    {f:1, h:6}, # g
    {f:9, g:8} # h
]
```

- Le `dict` d'adjacence peut être utilisé comme ci-dessus, avec la valeur en plus.

```
b in N[a] # Neighborhood membership
# True
len(N[f]) # Degree
#3
N[a][b] # Edge weight for (a, b)
#2
```

Représentation par dict : graphe valué (suite)

Dans cette représentation (par Dicts) :

- L'utilisation des dicts pour les graphes est intéressante dans la mesure où on utilise un ensemble (set).

De plus, cette représentation est compatible avec les anciennes versions de Python qui n'avait pas le type `set` (mais avaient ensembles par construction, avec '`{}`').

- Même si on n'a pas besoin de représenter un poids (la valeur) dans les noeuds, on peut continuer à utiliser les *dicts* et utiliser *None* ou toute autre valeur pour la pondération.

Représentation de graphes par dict d'ensemble

- A la représentation ci-dessus par `set`, `list` ou `dict` indexée par le numéro du noeud, on peut préférer une autre représentation plus souple est d'utiliser un `dict` où la clé peut être *hashée* (clé *hashable*) et où les valeurs seront des ensembles d'adjacents.
- Dans l'exemple ci-dessous, les noeuds sont des caractères :

```
N = {  
    'a': set('bcdef'),  
    'b': set('ce'),  
    'c': set('d'),  
    'd': set('e'),  
    'e': set('f'),  
    'f': set('cgh'),  
    'g': set('fh'),  
    'h': set('fg')  
}
```

- Notons que si on enlève les constructeurs `set(...)`, les adjacents d'un noeud serait une chaîne de caractère non mutable.
 - Ça marche aussi ! Le choix dépend de ce que l'on veut faire du graphe et comment le graphe nous est fourni (sous forme de texte ?).

Représentation par matrice d'adjacence

```
a, b, c, d, e, f, g, h = range(8)

#   a b c d e f g h

N = [[0,1,1,1,1,0,0],      # a
      [0,0,1,0,1,0,0,0],    # b
      [0,0,0,1,0,0,0,0],    # c
      [0,0,0,0,1,0,0,0],    # d
      [0,0,0,0,0,1,0,0],    # e
      [0,0,1,0,0,0,1,1],    # f
      [0,0,0,0,0,1,0,1],    # g
      [0,0,0,0,0,1,1,0]]    # h

N[a][b]      # Neighborhood membership
# 1
sum(N[f])   # Degree
# 3
```

- Ici, un '1' atteste la présence d'un arc et '0' son absence.
→ On peut bien entendu utiliser True / False.

Représentation par matrice d'adjacence (suite)

- Pour représenter un graphe valué (pondéré), on peut utiliser le format suivant où `inf` représente l'absence d'arc.
- Notons que les '0' représentent un cout (pondération) nul permettent de conserver le caractère réflexif de la fonction `arc` : il y a un arc de tout noeud à lui même de cout nul.
 - Ce mécanisme simplifie souvent les algorithmes.

```
a, b, c, d, e, f, g, h = range(8)
inf = float('inf')

#      a   b   c   d   e   f   g   h
W = [[0, 2, 1, 3, 9, 4, inf, inf],      # a
      [inf, 0, 4, inf, 3, inf, inf, inf],    # b
      [inf, inf, 0, 8, inf, inf, inf, inf],    # c
      [inf, inf, inf, 0, 7, inf, inf, inf],    # d
      [inf, inf, inf, inf, 0, 5, inf, inf],    # e
      [inf, inf, 2, inf, inf, 0, 2, 2],        # f
      [inf, inf, inf, inf, inf, 1, 0, 6],        # g
      [inf, inf, inf, inf, inf, 9, 8, 0]]       # h
```

Représentation par matrice d'adjacence (suite)

- Traces :

```
W[a][b] < inf      # Neighborhood membership
# True

W[c][e] < inf      # Neighborhood membership
# False

sum(1 for w in W[a] if w < inf) - 1 # Degree : le '-1' est pour les '0'
# 5
```

- La matrice ci-dessus simplifie l'accès aux pondérations, mais la vérification et la recherche du degré d'un noeud ou même l'itération sur les voisins sont réalisés différemment : on doit prendre en compte la valeur `inf`.

Parcours de graphes en Python

- **Exemple :** création d'un graphe orienté non pondéré à l'aide d'un **dictionnaire**.

A -> B

A -> C

B -> C

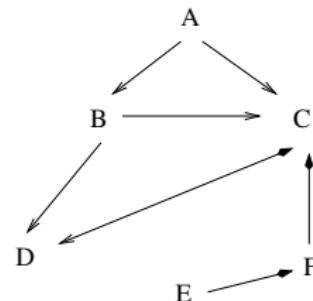
B -> D

C -> D

D -> C

E -> F

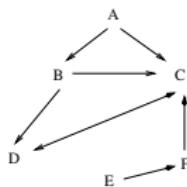
F -> C



```
graph = { 'A': [ 'B', 'C' ],
          'B': [ 'C', 'D' ],
          'C': [ 'D' ],
          'D': [ 'C' ],
          'E': [ 'F' ],
          'F': [ 'C' ]}
```

Parcours de graphes en Python (suite)

- Recherche de chemin :

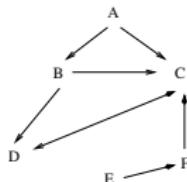


```
def chemin(graph, depart, arrivee , path=[]):
    path = path + [depart]
    if depart == arrivee :
        return path
    if not graph.has_key(depart):
        return None
    for noeud in graph[depart]:
        if noeud not in path:
            nouv_path = chemin(graph, noeud, arrivee , path)
            if nouv_path: return nouv_path
    return None

chemin(graph, 'A', 'D')
# ['A', 'B', 'C', 'D']
```

Parcours de graphes en Python (suite)

- Recherche de tous les chemins depuis un noeud :

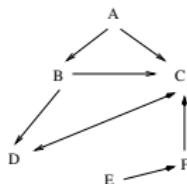


```
def tous_chemins(graph, depart, arrivee , path=[]):
    path = path + [depart]
    if depart == arrivee :
        return [path]
    if not graph.has_key(depart):
        return []
    paths = []
    for noeud in graph[depart]:
        if noeud not in path:
            nouv_paths = tous_chemins(graph, noeud, arrivee , path)
            for nouv_path in nouv_paths:
                paths.append(nouv_path)
    return paths

tous_chemins(graph, 'A', 'D')
# [[['A', 'B', 'C', 'D'], ['A', 'B', 'D'], ['A', 'C', 'D']]
```

Parcours de graphes en Python (suite)

- Chemin le plus court (en nombre d'arcs) par une stratégie B&B :



```
def chemin_le_plus_court(graph, depart, arrivee , path=[]):
    path = path + [depart]
    if depart == arrivee :
        return path
    if not graph.has_key(depart):
        return None
    shortest = None
    for noeud in graph[depart]:
        if noeud not in path:
            nouv_path = chemin_le_plus_court(graph, noeud, arrivee , path)
            if nouv_path:
                if not shortest or len(nouv_path) < len(shortest):
                    shortest = nouv_path
    return shortest

chemin_le_plus_court(graph, 'A', 'D')
# ['A', 'C', 'D']
```

Complément : librairies de Graphe de Python

- Quelques librairies de graphes pour Python :

- NetworkX : <http://networkx.lanl.gov>
- python-graph : <http://code.google.com/p/python-graph>
- Graphine : <https://gitorious.org/graphine/pages/Home>
- Graph-tool : <http://graph-tool.skewed.de>

Et aussi :

- Pygr (<https://github.com/cjlee112/pygr>)
- Gato, un toolbox d'animation de graphes (<http://gato.sourceforge.net>) ;
- PADS : une collection d'algorithmes de graphes
(<http://www.ics.uci.edu/~eppstein/PADS>).

Python et Graph_tool

- Les modules *NetworkX*, *graph_tool* et *igraph* sous Python proposent des graphes.
- Graph_tool est réalisé en interne en *C++* à l'aide de *Boost*. Il est difficile de l'installer sous Python. Ici, on voit juste un exemple.
- Voir aussi <https://wiki.python.org/moin/PythonGraphApi> pour les graphes sous Python.

```
from graph_tool.all import *
g = Graph()                      # Par défaut, un graphe orienté
ug = Graph(directed=False)        # Pour un graphe non orienté
```

- On peut changer un graphe orienté en non orienté et inversement à la volée à l'aide de la fonction *set_directed()*.
- On peut tester cette propriété du graphe par la fonction *is_directed()*.

```
ug = Graph()
ug.set_directed(False)
assert(ug.is_directed() == False)
```

Python et Graph_tool (suite)

- On ajoute des noeuds avec `add_vertex()`.

```
v1 = g.add_vertex()      # renvoie la description du noeud dans v1
v2 = g.add_vertex()
```

- On peut ajouter des arcs par `add_edge()`. On relie les noeuds `v1` et `v2` :

```
e = g.add_edge(v1, v2)
```

- On peut demander à dessiner un graphe :

```
graph_draw(g, vertex_text=g.vertex_index, vertex_font_size=18, output_size=(200, 200), output="two-nodes.png")
```

- On peut créer un graphe à partir d'un autre :

```
g1 = Graph()
# ... remplir g1
g2 = Graph(g1)           # g1 et g2 sont des copies
```

→ La copie est ici réelle (dite "copie profonde").

→ Ce ne sont pas deux références sur le même graphe.

Tab Mat

1

Introduction

- Plan général des actions TC
- Plan général du cours
- Organisation (de ce cours / chapitre)

2

Quelques algorithmes utilisés de nos jours

- De nos jours : Google page Ranking
- Facebook's News Feed
- OKCupid Date Matching
- NSA Data Collection
- IBM'CRUSH
- Autres algorithmes remarquables
- Robotique

3

Machines et Algorithmes

- Structure des données
- Classification des algorithmes
- Anatomie d'une instruction
- Pile et contexte machine

4

Formalismes et Langages de programmation

- Propriétés des algorithmes
- Décomposition des SDs

5

Python par exemples

- Quelques éléments du langage Python
- Les tours de Hanoi
- Palindrome
- Recherche du mot le plus long dans une chaîne
- Chiffres Romains

Tab Mat (suite)

- Fréquence des lettres dans un texte

6 Types de données

7 Annexes Cours 1

- Un exemple de réduction
- Entraînées d'une machine
- Anatomie du contexte d'un processus
- Place et rôle de l'algorithmique

8 La récursivité

- La récursivité dans la 'nature'
- Exemples de Récursivité
- Remarques sur la récursivité et Python
- Types de données récursifs
- Durée de vie des variables
- Décomposition récurrente
- Exemple : médiane
- Récursivité et Retours arrières
- Résumé de la récursivité

9 Récurrence et Preuve

- Preuve par l'Induction Mathématique
- Exemple 2
- Exemple Postier
- Induction Forte

10 Induction Constructive

11 Annexes Cours 2

- Récursivité : Le compte est bon

Tab Mat (suite)

- Un autre ex. de décomposition récursive
- Décomposition : Dragons récursifs et Tkinter
- Un autre ex.e d'induction constructive

12

Complexité

- Sensibilité à la machine
- Fonction Oméga
- Ordre des fonctions
- Calcul de complexité
- Exemple de Calcul de complexité

13

Complément à la Complexité

14

Les arbres

- Quelques définitions
- Représentation des arbres
- Représentation d'Arbres binaires
- Parcours d'arbre binaire
- Arbres binaires : un exemple Python
- Arbres : représentation par listes imbriquées
- Python : Fonctions usuelles sur arbres binaires
- Arbres : quelques algorithmes
- ABOH en Python
- Exercice sur les Arbres

15

Pile et File en Python

16

Complément : alternatives aux Pile et File en Python

17

TAS ou Heap

- TAS en Python

Tab Mat (suite)

18

En savoir plus : Transformation de la récursivité

- Récursivité terminale
- Récursivité non terminale
- Récursivité non terminale : exemples
- Exemples plus élaborés
- Addendum : Parcours itératif d'arbre binaire

19

Exceptions : Traitement des erreurs, ruptures

- Exceptions Python

20

Graphes

- Représentation abstraite des graphes
- Parcours général de graphes

21

Graphes en Python

- Représentation des graphes en Python
- Parcours de graphes en Python
- Les librairies de Graphe de Python
- Python et le module graph_tool