

Algorithmique et structures de données

Cours INF TC1

Romain Vuillemot
`romain.vuillemot@ec-lyon.fr`

Département Math-Info
École Centrale de Lyon

2021-2022

Sommaire

Introduction

- Exemple d'algorithme
- Objectifs du module INF-TC1
- Ressources

Principes d'algorithmique

- Définitions
- Schémas d'algorithmes
- Exemples d'algorithmes

Principes généraux

- Récursivité
- Complexité

Types et structures de données

- Tables de Hachage
- Piles et files
- Files de priorité
- Listes liées

Séances de cours de INF TC1

- ▶ Cours 1: introduction aux algorithmiques et structures de données
Hash maps, Piles, Files, Listes de priorité, Listes chaînées
- ▶ Cours 2: stratégies de programmation
Arbres, Parcours Profondeur/Largeur, Diviser pour Reignier, Glouton
- ▶ Cours 3: structure de données avancées
Graphes, Arbres couvrant, Parcours de Graphe, Recherche de Chemin
- ▶ Cours 4: algorithmes avancés

Introduction aux Algorithmes

Introduction

Cours 1

Objetifs du cours :

- ▶ Notions de récursivité / complexité
- ▶ Structures de base (listes, tableaux, dictionnaires, sets)
- ▶ Tables de hachage
- ▶ Piles / Files
- ▶ Listes de priorité
- ▶ Listes chaînées

Lors des TDs nous verrons leur mise en oeuvre en fonction de cas d'études. En particulier les outils pour justifier vos choix ! (en termes de performance, complexité, etc.)

Sommaire

Introduction

- Exemple d'algorithme
- Objectifs du module INF-TC1
- Ressources

Principes d'algorithmique

- Définitions
- Schémas d'algorithmes
- Exemples d'algorithmes

Principes généraux

- Récursivité
- Complexité

Types et structures de données

- Tables de Hachage
- Piles et files
- Files de priorité
- Listes liées

Qu'est-ce qu'un algorithme ?

Question :

Quel exemple d'algorithme connaissez-vous ?

Comment fonctionnent-ils ?

¹<https://mathematical-tours.github.io/algorithms/>

Qu'est-ce qu'un algorithme ?

Question :

Quel exemple d'algorithme connaissez-vous ?

Comment fonctionnent-ils ?

Font-ils des erreurs ? Ont-ils des biais ?

¹<https://mathematical-tours.github.io/algorithms/>

Qu'est-ce qu'un algorithme ?

Question :

Quel exemple d'algorithme connaissez-vous ?

Comment fonctionnent-ils ?

Font-ils des erreurs ? Ont-ils des biais ?

Les algorithmes sont désormais partout dans notre quotidien. Les premiers algorithmes très anciens¹ (le mot dérivé du nom Mūsā al-Khwārizmī d'un mathématicien Perse du 9ème siècle).

Par exemple l'algorithme d'Euclide :

1. diviser a par b, on obtient le reste r
2. remplacer a par b
3. remplacer b par a
4. continuer tant que c'est possible, sinon on obtient le pgcd

¹<https://mathematical-tours.github.io/algorithms/>

Un algorithme (classique et important)

Question :

Comment selon-vous Google classifie les pages web ?

² *The Anatomy of a Large-Scale Hypertextual Web Search Engine* <http://infolab.stanford.edu/pub/papers/google.pdf>

Un algorithme (classique et important)

Question :

Comment selon-vous Google classifie les pages web ?

Algorithme de PAGE RANK

- ▶ Introduit par Brin et Page (1999)²
- ▶ Permet de quantifier la popularité d'une page web selon ses liens
- ▶ Et classer les pages web lors de résultats de recherche (Google)

Formule itérative qui calcule le Page Rank pour une itération en fonction des pages qui pointent vers cette page et divisé par le page-rank des pages sortantes

² The Anatomy of a Large-Scale Hypertextual Web Search Engine <http://infolab.stanford.edu/pub/papers/google.pdf>

Introduction

Algorithme du Page Rank

Formule du PAGE RANK :

$$PR_{t+1}(P_i) = \sum_{P_j} \frac{PR_t(P_j)}{C(P_j)}$$

Avec C le nombre de noeuds sortant de P_j .

Introduction

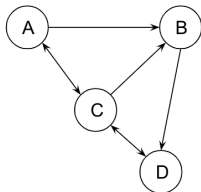
Algorithme du Page Rank

Formule du PAGE RANK :

$$PR_{t+1}(P_i) = \sum_{P_j} \frac{PR_t(P_j)}{C(P_j)}$$

Avec C le nombre de noeuds sortant de P_j .

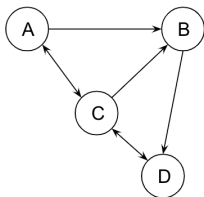
Exemple de modélisation de pages web avec un graphe orienté



	Iteration 0	Iteration 1	Iteration 2	Page Rank
A				
B				
C				
D				

Introduction

Algorithme classique

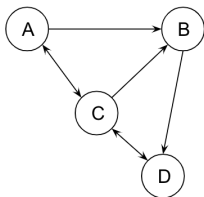


	Iteration 0	Iteration 1	Iteration 2	Page Rank
A	1/4	1/12	1.5/12	1
B	1/4	2.5/12	2/12	2
C	1/4	4.5/12	4.5/12	4
D	1/4	4/12	4/12	3

- Initialisation avec 1/4 (choix arbitraire mais pour que le PR total soit de 1)
- Le nœud le plus populaire est.. C !

Introduction

Algorithme classique



	Iteration 0	Iteration 1	Iteration 2	Page Rank
A	1/4	1/12	1.5/12	1
B	1/4	2.5/12	2/12	2
C	1/4	4.5/12	4.5/12	4
D	1/4	4/12	4/12	3

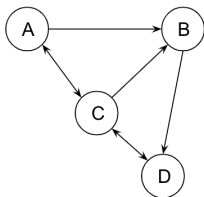
- Initialisation avec 1/4 (choix arbitraire mais pour que le PR total soit de 1)
- Le nœud le plus populaire est.. C !

(Cet exemple aborde des concepts que nous verrons dans ce cours : structure de graphe, exécution récursive, parcours de graphe, etc.)

Dans la réalité, l'implémentation de l'algorithme beaucoup plus complexe (personnalisation, ciblage publicitaire, etc.), et dépend de sa vue du web (grâce aux *crawlers*).

Introduction

Algorithme classique

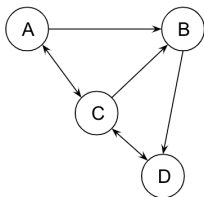


	Iteration 0	Iteration 1	Iteration 2	Page Rank
A	1/4	1/12	1.5/12	1
B	1/4	2.5/12	2/12	2
C	1/4	4.5/12	4.5/12	4
D	1/4	4/12	4/12	3

- Initialisation avec 1/4 (choix arbitraire mais pour que le PR total soit de 1)
- Le nœud le plus populaire est.. C !
- NB: D est populaire car C (populaire) pointe vers lui !

Introduction

Algorithme classique



	Iteration 0	Iteration 1	Iteration 2	Page Rank
A	1/4	1/12	1.5/12	1
B	1/4	2.5/12	2/12	2
C	1/4	4.5/12	4.5/12	4
D	1/4	4/12	4/12	3

- ▶ Initialisation avec 1/4 (choix arbitraire mais pour que le PR total soit de 1)
- ▶ Le nœud le plus populaire est.. C !
- ▶ NB: D est populaire car C (populaire) pointe vers lui !

(Cet exemple aborde des concepts que nous verrons dans ce cours : structure de graphe, exécution récursive, etc.)

Dans la réalité, l'implémentation de l'algorithme beaucoup plus complexe (personnalisation, ciblage publicitaire, etc.), et dépend de sa vue du web (grâce aux *crawlers*).

Objectifs du module INF-TC1

Acquis (Savoir/Cours) :

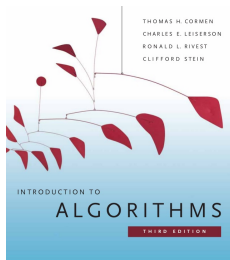
- ▶ Connaître les principaux algorithmes en informatique
- ▶ Principales structures de données, méthodes de programmation
- ▶ Applications à des problèmes de recherche, tri, et apprentissage

Acquis (Savoir/Cours) :

- ▶ Connaître les principaux algorithmes en informatique
- ▶ Principales structures de données, méthodes de programmation
- ▶ Applications à des problèmes de recherche, tri, et apprentissage

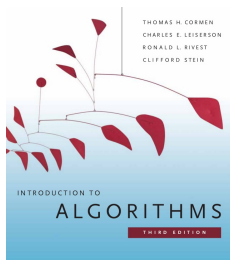
Compétences (Savoir-Faire/TD/Projets) :

- ▶ Étant donné un problème, choisir et appliquer le bon algorithme
- ▶ Écrire un algorithme sous ses différentes formes (pseudo, code, graphique) pour le communiquer
- ▶ Réaliser ses propres structures de données adaptées au problème
- ▶ Justifier les choix réalisés par analyse théorique (calcul de complexité) et empirique (analyse de performance)



Introduction to Algorithms

Thomas H. Cormen. Charles E. Leiserson. Ronald L. Rivest. Clifford Stein 3ème édition MIT Press (2010)



Introduction to Algorithms

Thomas H. Cormen. Charles E. Leiserson. Ronald L. Rivest. Clifford Stein 3ème édition MIT Press (2010)

Autres livres:

- ▶ Algorithms, 4th Edition. *Robert Sedgewick, Kevin Wayne*
- ▶ The Art of Computer Programming. *Donald Knuth*
- ▶ Algorithms in a Nutshell. *George Heineman*

Sommaire

Introduction

- Exemple d'algorithme
- Objectifs du module INF-TC1
- Ressources

Principes d'algorithmique

- Définitions
- Schémas d'algorithmes
- Exemples d'algorithmes

Principes généraux

- Récursivité
- Complexité

Types et structures de données

- Tables de Hachage
- Piles et files
- Files de priorité
- Listes liées

Principes d'algorithmique

Définition d'algorithme

Un algorithme est un **ensemble d'instructions, non-ambigue, permettant de résoudre un problème.**

Principes d'algorithmique

Définition d'algorithme

Un algorithme est un **ensemble d'instructions, non-ambigue, permettant de résoudre un problème.**

Autrement dit une suite **totalement définie** d'opérations et de leurs enchaînements ayant pour but le traitement (et la transformation) d'informations.

Principes d'algorithmique

Définition d'algorithme

Un algorithme est un **ensemble d'instructions, non-ambigue, permettant de résoudre un problème.**

Autrement dit une suite **totalement définie** d'opérations et de leurs enchaînements ayant pour but le traitement (et la transformation) d'informations.

Un algorithme possède plusieurs propriétés :

- ▶ Communicable
- ▶ Efficace
- ▶ Complet, termine et correct
- ▶ Déterministe

NB: nous verrons dans la dernière partie du cours des algorithmes qui ne vérifient pas forcément ces propriétés. Par exemple algorithmes non-déterministes, heuristiques, etc.

Principes d'algorithmique

Communiquer un algorithme

Il existe différentes façon d'exprimer un algorithme, tout dépendant du contexte et du niveau de *formalisation* nécessaire.

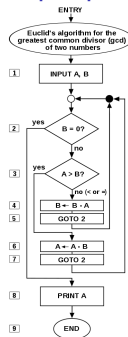
Pseudo-code

diviser a par b, reste r
remplacer a par b
remplacer b par a
continuer tant que
c'est possible,
sinon on obtient le pgcd.

Implémentation

```
def pgcd(a,b):  
    while b  $\neq$  0:  
        a, b = b, a%b  
    return a
```

Graphique



Principes d'algorithmique

Communiquer un algorithme

Il existe différentes façon d'exprimer un algorithme, tout dépendant du contexte et du niveau de *formalisation* nécessaire.

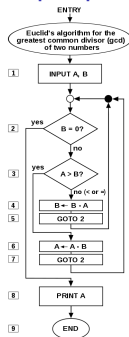
Pseudo-code

diviser a par b, reste r
remplacer a par b
remplacer b par a
continuer tant que
c'est possible,
sinon on obtient le pgcd.

Implémentation

```
def pgcd(a,b):  
    while b > 0:  
        a, b = b, a%b  
    return a
```

Graphique



Question :

Indiquez les avantages/inconvénients de chaque représentation ?
Connaissez-vous d'autres représentations ?

Principes d'algorithmique

Communiquer un algorithme

Il existe différentes façon d'exprimer un algorithme, tout dépendant du contexte et du niveau de *formalisation* nécessaire.

- ▶ La **représentation graphique** est plus abordable et permet d'avoir un aperçu global et éventuellement détecter des erreurs, patterns, etc. Car l'humain a de meilleurs capacités de perception dans l'espace visuel plutôt que le texte.
- ▶ Le **pseudo-language** a la particularité d'être flexible, proche du langage humain mais aussi informatique, indépendant d'un langage de programmation. Par contre souvent défini de manière ambiguë et demande un effort supplémentaire pour l'implémentation.
- ▶ Enfin **l'implémentation (ex: Python)** a l'avantage d'être immédiatement testable. Par contre est très contraignante (doit être juste), parfois difficile à lire si on ne connaît pas le langage. Dépend aussi du programmeur.

Principes d'algorithmique

Différence avec un programme



"I will, in fact, claim that the difference between a **bad** programmer and a **good** one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships."

— Linus Torvalds

Principes d'algorithmique

Différence avec un programme

Quelle que soit la représentation, un algorithme doit in fine être traduit dans un **langage de programmation** et donc dans un programme.

- ▶ La représentation (ou parfois même traduction) en langage de programmation n'est pas réciproque : **tout programme n'est pas un algorithme.**
- ▶ Par exemple les programmes réactifs (gestion des entrées / sorties), ou qui contiennent des animations, ne se terminent pas car toujours en attente de commande. Ils ne constituent pas à proprement parler des algorithmes.
- ▶ D'autres programmes sont aussi dépendant de paramètres extérieures (ex: réseau, disponibilité de données, etc.). Le périmètre algorithmique ici n'est pas très bien défini.

Principes d'algorithmique

Différence avec un programme

Quelle que soit la représentation, un algorithme doit in fine être traduit dans un **langage de programmation** et donc dans un programme.



1. Problème à résoudre (cahier des charges)
2. Spécification (formalisation du problème)
3. **Algorithme(s)** et structures de données
4. Réalisation (à l'aide d'un langage de programmation)
5. Résultats

Principes d'algorithmique

Un exemple d'algorithme

Tri d'une liste. Ordonner les nombres suivants :

26 54 93 17 77 31 44 55 20

Principes d'algorithmique

Un exemple d'algorithme

Tri d'une liste. Ordonner les nombres suivants :

26 54 93 17 77 31 44 55 20

- ▶ Quelle structure de données utiliser pour les entrées ?
- ▶ Quel est le format de sortie ?
- ▶ Quelles sont les autres contraintes ?
- ▶ ...
- ▶ Connaissez-vous une solution ?

Principes d'algorithmique

Un exemple d'algorithme

Tri d'une liste. Ordonner les nombres suivants :

26 54 93 17 77 31 44 55 20

- ▶ Quelle structure de données utiliser pour les entrées ?
- ▶ Quel est le format de sortie ?
- ▶ Quelles sont les autres contraintes ?
- ▶ ...
- ▶ Connaissez-vous une solution ?

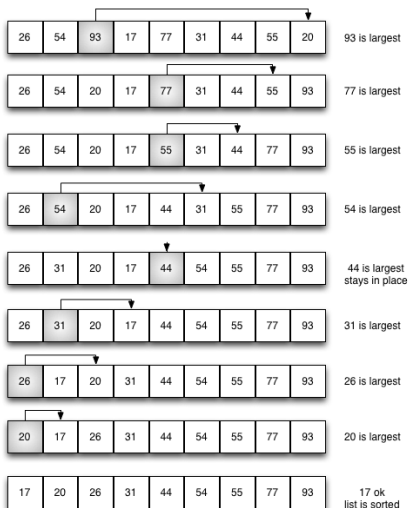
Ex: Tri par sélection (une liste en entrée et en sortie)

- 1.
2. Parcourt toute la liste et place le plus grand à la fin
3. Répète le processus en plaçant le plus grand en avant-dernier
4. Jusqu'à ce que la liste soit vide

Principes d'algorithmique

Un exemple d'algorithme

Tri par sélection (Selection Sort) parcourt toute la liste pour identifier le maximum (minimum), le place à la fin (au début), et recommence.



Tri par sélection (Selection Sort)

```
def selectionSort(alist):  
    for i in range(0, len(l)):  
        min = i  
        for j in range(i+1, len(l)):  
            if(l[j] < l[min]):  
                min = j  
        tmp = l[i]  
        l[i] = l[min]  
        l[min] = tmp  
    return l  
if __name__=="__main__":  
    liste = [54,26,93,17,77,31,44,55,20]  
    selectionSort(liste)  
    print(liste) # [17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Il s'agit d'une des méthodes de tri les plus simple et naïve.. et donc la méthode la moins performante. Complexité de $\mathcal{O}(n^2)$

Principes d'algorithmique

Efficacité d'un algorithme

Un algorithme est dit **efficace** si il minimise la consommation de ressource nécessaires pour le réaliser.

Principes d'algorithmique

Efficacité d'un algorithme

Un algorithme est dit **efficace** si il minimise la consommation de ressource nécessaires pour le réaliser.

L'efficacité est donc relative à différents critères (valeur que l'on souhaite mesurer) qu'il est nécessaire de calculer (théoriquement) ou de mesurer (empiriquement), afin de comprendre ce qu'il se passe.

A noter qu'il est alors nécessaire de prendre de grandes valeurs de n pour avoir un comportement représentatif.

Parmis ces critères :

- ▶ Temps d'exécution
- ▶ **Complexité**
- ▶ Espace mémoire nécessaire
- ▶ Espace disque
- ▶ Etc.

La complexité permet d'être indépendant de la technologie utilisée (langage, ordinateur, compilateur, etc.)

Exemple. En génomique, il est fréquent de comparer deux séquences (de gènes) de longueur N et M (ex : TAG CAC et TGC TTG)

- ▶ Le nombre de comparaisons est $N \times M$
- ▶ Si la taille des séquences double, alors le nombre de comparaisons... quadruple !
- ▶ $(2 \times N) \times (2 \times M) = 4 \times (N \times M)$
- ▶ Maintenant nous souhaitons aligner 3 séquences alors N^3

Principes d'algorithmique

Efficacité d'un algorithme

Exemple. En génomique, il est fréquent de comparer deux séquences (de gènes) de longueur N et M (ex : TAG CAC et TGC TTG)

- ▶ Le nombre de comparaisons est $N \times M$
- ▶ Si la taille des séquences double, alors le nombre de comparaisons... quadruple !
- ▶ $(2 \times N) \times (2 \times M) = 4 \times (N \times M)$
- ▶ Maintenant nous souhaitons aligner 3 séquences alors N^3

En pratique il sera difficile (on veut éviter d'aligner des séquences au delà de 2 séquences)

→ De même pour de longues séquences

→ Il est donc nécessaire d'avoir un algorithme efficace (dans le cas de la comparaison de séquences, voir l'algorithme BLAST³).

³<https://en.wikipedia.org/wiki/BLAST>

Principes d'algorithmique

Complet, termine et correct

Les autres qualités d'un algorithme (au delà d'être simple et compréhensible) :

Un algorithme doit être **complet**, c'est à dire pour un problème donné il fourni une solution pour chacune des entrées.

Un algorithme doit se **terminer** dans un temps fini.

Un algorithme doit être **correct** et terminer en fournissant un résultat qui est la solution au problème qu'il doit résoudre.

Principes d'algorithmique

Complet, termine et correct

Les autres qualités d'un algorithme (au delà d'être simple et compréhensible) :

Un algorithme doit être **complet**, c'est à dire pour un problème donné il fourni une solution pour chacune des entrées.

Un algorithme doit se **terminer** dans un temps fini.

Un algorithme doit être **correct** et terminer en fournissant un résultat qui est la solution au problème qu'il doit résoudre.

→ Tout cela est très difficile à prouver (preuve formelle, ..) !

Principes d'algorithmique

Schémas d'algorithmes

Un algorithme possède un **schéma** qui est une manière de les classer selon ses propriétés.

Principes d'algorithmique

Schémas d'algorithmes

Un algorithme possède un **schéma** qui est une manière de les classer selon ses propriétés.

- ▶ Il existe en effet plusieurs façon de concevoir des algorithmes en fonction soit de contraintes de performances, soit en fonction du style de la structure
- ▶ Il n'existe pas un seul algorithme unique pour un problème donnée

Exemples de schémas (principaux):

- ▶ Par finalité
- ▶ Par implémentation (ex: **récurtivité**, fonctionnelle, etc.)
- ▶ Par **paradigme de conception** (Diviser-pour-régner, etc.)
- ▶ Par **complexité**

Principes d'algorithmique

Ressources d'un ordinateur

Les algorithmes doivent optimiser les ressources d'un ordinateur.

Quelles sont ces ressources ?

Principes d'algorithmique

Ressources d'un ordinateur

Les algorithmes doivent optimiser les ressources d'un ordinateur.

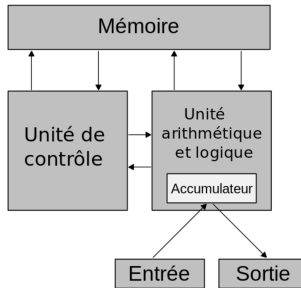
Quelles sont ces ressources ?

- ▶ Puissance de calcul
- ▶ Mémoire (vive, stockage, cache)
- ▶ Stockage externe
- ▶ Débit interne (bus)
- ▶ Débit externe (réseau)
- ▶ Adressage de valeurs/variables
- ▶ Etc.

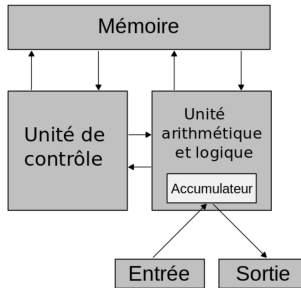
Principes d'algorithmique

Comment sont stockées les données ?

Comment sont stockées les données ?



Comment sont stockées les données ?

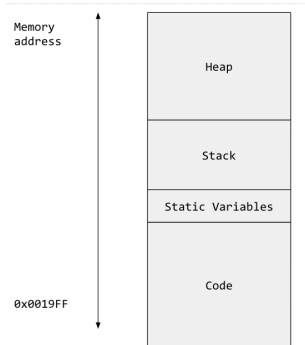


1. **Unité de traitement** : pour effectuer les opérations de base
2. **Unité de contrôle** : permet d'enchaîner les opérations à réaliser
3. **Unité de mémoire** : contient données et programme, et permettra à l'unité de contrôle de savoir quel calcul réaliser sur ces données.
 - 3.1 Mémoire volatile
 - 3.2 Mémoire permanente
4. **Entrées et sorties** : afin de communiquer avec l'environnement

Principes d'algorithmique

Comment sont stockées les données ?

La mémoire est divisée en plusieurs zones :

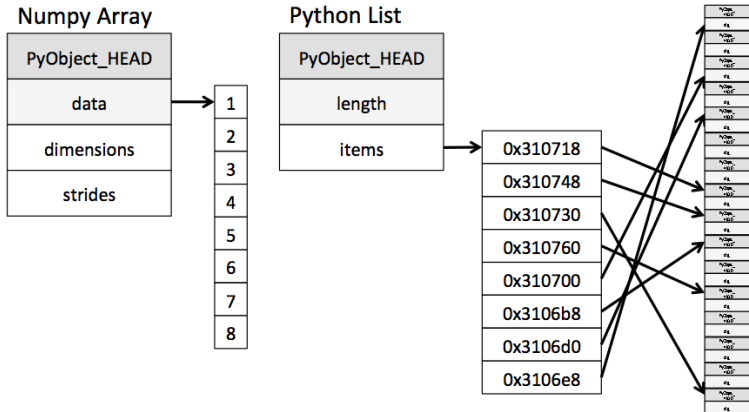


- ▶ **Heap** : zone mémoire réservée pour l'allocation dynamique
- ▶ **Stack** : stocke les références aux variables et les fonctions
- ▶ **Static variables** : variables des fonctions ou variables des fonctions
- ▶ **Code** : stocke les instructions, fonctions, sous forme de programme compilé

<https://docs.python.org/3/c-api/memory.html>

Principes d'algorithmique

Adressage de données



Les données sont soit stockées de manière séquentielle (grand tableau) ou bien de manière aléatoire si il s'agit d'une liste de références.

<https://docs.python.org/3/library/functions.html#id>

Principes d'algorithmique

Adressage de données

Testez vous-même l'affichage d'adresses de variables en Python :

```
x = 4
y = 4
w = 9999
v = 9999
a = 12345678
b = 12345678
print(hex(id(x)))
print(hex(id(y)))
print(hex(id(w)))
print(hex(id(v)))
print(hex(id(a)))
print(hex(id(b)))
```

Principes d'algorithmique

Adressage de données

Testez vous-même l'affichage d'adresses de variables en Python :

```
x = 4
y = 4
w = 9999
v = 9999
a = 12345678
b = 12345678

print(hex(id(x)))
print(hex(id(y)))
print(hex(id(w)))
print(hex(id(v)))
print(hex(id(a)))
print(hex(id(b)))
```

```
>>> print(hex(id(x)))
0x7ff3ff609760
>>> print(hex(id(y)))
0x7ff3ff609760
>>> print(hex(id(w)))
0x7ff3ff409f90
>>> print(hex(id(v)))
0x7ff3ff409f60
>>> print(hex(id(a)))
0x7ff3ff409f48
>>> print(hex(id(b)))
0x7ff3ff409f18
```

Certaines variables ont une place constante en mémoire (int de 0 à 256)

Principes d'algorithmique

Exemples d'algorithmes (BONUS)

Exemple. Un algorithme identifie les mots qui se répètent n fois : quelle structure de données est utilisée? Est-il juste? Complet? Communicable?

```
def countWords(stri, n):  
    m = dict()  
    for i in range(n):  
        m[stri[i]] = m.get(stri[i],0) + 1  
    res = 0  
    for i in m.values():  
        if i == 2:  
            res += 1  
    return res  
  
if __name__=="__main__":  
    # Driver code  
    s = [ "hate", "love", "peace", "love", "peace", "hate",  
          "love", "peace", "love", "peace" ]  
    n = len(s)  
    print(countWords(s, n))
```

Principes d'algorithmique

Exemples d'algorithmes (BONUS)

Exemple. Un algorithme identifie les mots qui se répètent n fois : quelle structure de données est utilisée? Est-il juste? Complet? Communicable?

```
def countWords(stri, n):  
    m = dict()  
    for i in range(n):  
        m[stri[i]] = m.get(stri[i],0) + 1  
    res = 0  
    for i in m.values():  
        if i == 2:  
            res += 1  
    return res  
  
if __name__=="__main__":  
    # Driver code  
    s = [ "hate", "love", "peace", "love", "peace", "hate",  
          "love", "peace", "love", "peace" ]  
    n = len(s)  
    print(countWords(s, n))
```

Une liste de mots est utilisé et l'algorithme est juste, complet et communicable.

Principes d'algorithmique

Exemples d'algorithmes (BONUS)

Exemple. Un algorithme qui étant donné N dés avec chacun M côtés, numérotés de 1 à M , trouver le nombre de possibilités d'obtenir la somme X . X étant la somme des valeurs une fois les dés jetés. La description ci-dessous est-elle un algorithme ?

Par exemple pour $M = 6$, $N = 3$, $X = 12$
on attend comme sortie 25

Principes d'algorithmique

Exemples d'algorithmes (BONUS)

Exemple. Un algorithme qui étant donné N dés avec chacun M côtés, numérotés de 1 à M , trouver le nombre de possibilités d'obtenir la somme X . X étant la somme des valeurs une fois les dés jetés. La description ci-dessous est-elle un algorithme ?

Par exemple pour $M = 6$, $N = 3$, $X = 12$
on attend comme sortie 25

Non.. voici un algorithme avec 3 variables en entrée et 1 variable en sortie:

```
m,n,x = list(map(int,input().split()))
dp=[[0]*(x+1) for i in range(n+1)]
for i in range(1,min(x,m)+1):
    dp[1][i]=1
for i in range(2,n+1):
    for j in range(1,x+1):
        for k in range(1,min(j,m+1)):
            dp[i][j]+=dp[i-1][j-k]
print(dp[n][x])
```

Sommaire

Introduction

- Exemple d'algorithme
- Objectifs du module INF-TC1
- Ressources

Principes d'algorithmique

- Définitions
- Schémas d'algorithmes
- Exemples d'algorithmes

Principes généraux

- Réversivité
- Complexité

Types et structures de données

- Tables de Hachage
- Piles et files
- Files de priorité
- Listes liées

Récurtivité

Définition de la récursivité

Une fonction est dite **récursive** si, de manière générale elle s'appelle elle-même. En particulier d'un cas simple (dit cas *terminal*) et d'instructions permettant d'atteindre le cas terminal (un ou plusieurs appels à elle-même).

Définition de la récursivité

Une fonction est dite **récursive** si, de manière générale elle s'appelle elle-même. En particulier d'un cas simple (dit cas *terminal*) et d'instructions permettant d'atteindre le cas terminal (un ou plusieurs appels à elle-même).

Remarques :

- ▶ Une fonction récursive a toujours au moins un paramètre
- ▶ On évite cependant de passer trop de paramètres (et surtout inutiles)

Principes généraux

Exemples de récursivité

Question :

Connaissez-vous des exemples de phénomènes récursifs?

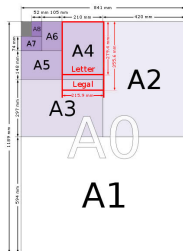
Principes généraux

Exemples de récursivité

Question :

Connaissez-vous des exemples de phénomènes récursifs?

Construction récursive



- ▶ 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 = $\frac{1}{2} \times$ Longueur de A_{i-1}

https://en.wikipedia.org/wiki/Paper_size

Types de données récurtifs

Certains **types de données** sont par nature réursive, à savoir un sous-ensemble de ce type de données possède le même type de données :

- ▶ Listes
- ▶ Chaînes de caractères (tableaux)
- ▶ Arbres binaires
- ▶ Listes chaînées
- ▶ Etc.

Récursivité

Types de données récurifs

Certains **types de données** sont par nature réursive, à savoir un sous-ensemble de ce type de données possède le même type de données :

- ▶ Listes
- ▶ Chaînes de caractères (tableaux)
- ▶ Arbres binaires
- ▶ Listes chaînées
- ▶ Etc.

Exemple. Recherche du maximum dans une liste L:

```
def maximum(L):  
    if len(L) == 1 : return L[0]  
    return max(L[0], maximum(L[1:]))
```

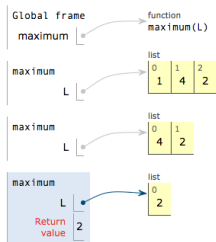
- ▶ La fonction `max()` donne le plus grand parmi ses 2 paramètres.
- ▶ Le cas `L = []` non traité (une exception `IndexError` est déclenchée).

Récurtivité

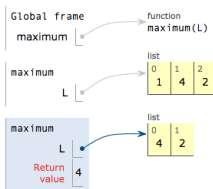
Exécution du code

Représentation des états **successifs** de la fonction `maximum` précédente :

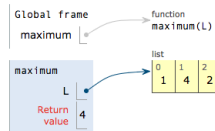
Cas terminal



Dépilement



Solution



Version interactive des appels récurrents

<http://www.pythontutor.com/visualize.html#mode=edit>

En cas d'appels trop nombreux *maximum recursion depth exceeded error* est déclenchée. Il est possible d'anticiper cette erreur en ne dépassant pas la limite `sys.getrecursionlimit()` ou bien en changeant celle-ci `sys.setrecursionlimit(10**6)`

Principes généraux

Réversivité vs Itérativité

- ▶ La réversivité peut être naturelle pour certains problèmes (ou structures de données qui sont elles-mêmes réversifs, par exemple)
- ▶ Cependant parfois il sera demandé d'explicitement comment passer de l'un a l'autre.
- ▶ Le changement résultera en des fonctions identiques en termes d'entrées et de sorties; mais avec un schéma algorithmique différent.

Principes généraux

Réversivité vs Itérativité

- ▶ La réversivité peut être naturelle pour certains problèmes (ou structures de données qui sont elles-mêmes réversifs, par exemple)
- ▶ Cependant parfois il sera demandé d'explicitement comment passer de l'un a l'autre.
- ▶ Le changement résultera en des fonctions identiques en termes d'entrées et de sorties; mais avec un schéma algorithmique différent.

Exemple. Écrire une fonction `factoriel` qui calcule le factoriel d'un entier. Donnez la forme itérative et réversive.

Principes généraux

Réversivité vs Itérativité

- ▶ La réversivité peut être naturelle pour certains problèmes (ou structures de données qui sont elles-mêmes réversifs, par exemple)
- ▶ Cependant parfois il sera demandé d'explicitement comment passer de l'un à l'autre.
- ▶ Le changement résultera en des fonctions identiques en termes d'entrées et de sorties; mais avec un schéma algorithmique différent.

Exemple. Écrire une fonction `factoriel` qui calcule le factoriel d'un entier. Donnez la forme itérative et réversive.

Forme itérative

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

Forme réversive

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

Principes généraux

Récurtivité vs Itérativité

Exemple. Écrire une fonction `palindrome` qui indique si un mot peut être lu de manière identique dans les deux sens. Donnez la forme itérative et récursive.

```
>>> print(palindrome("laval"))
True
>>> print(palindrome("toto"))
False
```

Principes généraux

Réversibilité vs Itérativité

Exemple. Écrire une fonction `palindrome` qui indique si un mot peut être lu de manière identique dans les deux sens. Donnez la forme itérative et récursive.

```
>>> print(palindrome("laval"))
True
>>> print(palindrome("toto"))
False
```

Forme itérative

```
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);
```

Forme 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])
```

Réversivité

Terminale et non-terminale

Si le dernier appel est l'appel récursif, on parle de réversivité **terminale**.
Sinon, il s'agit de réversivité **non-terminale**.

Réversivité

Terminale et non-terminale

Si le dernier appel est l'appel récursif, on parle de récursivité **terminale**.
Sinon, il s'agit de récursivité **non-terminale**.

```
def ecrire_desc(n):  
    if n > 0:  
        print(" " * n, "  n= " , n , " avant l'appel récursif")  
        ecrire_desc (n - 1)  
  
    if __name__=="__main__":  
        ecrire_desc(3)  
  
>>>      n= 3 avant l' appel récursif  
>>>      n= 2 avant l' appel récursif  
>>> n= 1 avant l' appel récursif
```

Réversivité

Terminale et non-terminale

Si le dernier appel est l'appel récursif, on parle de réversivité **terminale**.
Sinon, il s'agit de réversivité **non-terminale**.

```
def ecrire_desc(n):  
    if n > 0:  
        print(" " * n, "  n= " , n , " avant l'appel récursif")  
        ecrire_desc (n - 1)  
  
    if __name__=="__main__":  
        ecrire_desc(3)  
  
>>>      n= 3 avant l' appel récursif  
>>>      n= 2 avant l' appel récursif  
>>> n= 1 avant l' appel récursif
```

```
def ecrire_asc(n) :  
    if n > 0:  
        ecrire_asc(n - 1)  
        print(" " * n, "  n= " , n , " après l'appel récursif")  
  
    ecrire_asc(3)  
  
>>> n= 1 après l' appel récursif  
>>>      n= 2 après l' appel récursif  
>>>      n= 3 après l' appel récursif
```

Réversivité

Terminale et non-terminale

Si le dernier appel est l'appel récursif, on parle de récursivité **terminale**.
Sinon, il s'agit de récursivité **non-terminale**.

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

if __name__=="__main__":
    ecrire_desc(3)

>>>      n= 3 avant l' appel récursif
>>>      n= 2 avant l' appel récursif
>>> n= 1 avant l' appel récursif
```

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

    ecrire_asc(3)

>>> n= 1 après l' appel récursif
>>>      n= 2 après l' appel récursif
>>>      n= 3 après l' appel récursif
```

Impact le résultat ET la durée de vie des variables !

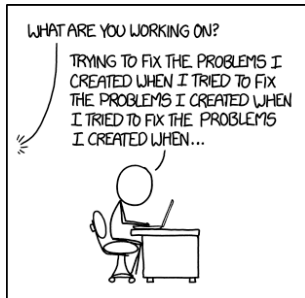
Réversivité

Décomposition récursive

Une **décomposition récursive** consiste à résoudre le problème dans le cas général en se ramenant aux cas particuliers où la solution est simple.

1. Trouver au moins un cas connu
2. Tenter de ramener le problème vers ce cas
3. Soit le cas est atteint, soit une valeur proche et recommencer

Bien vérifier que la récursion se termine⁴ ! (et est correcte..)



⁴<https://www.xkcd.com/1739/>

Principes généraux

Schémas de transformation

Il existe quelques schémas de transformation récursif vers itératif. Voici la démarche (générale) :

1. Écrire un algorithme récursif
2. Le tester, valider, prouver (dans les applications critiques)
3. Lui appliquer les techniques de transformation pour obtenir une version itérative

Principes généraux

Schémas de transformation

Il existe quelques schémas de transformation récursif vers itératif. Voici la démarche (générale) :

1. Écrire un algorithme récursif
2. Le tester, valider, prouver (dans les applications critiques)
3. Lui appliquer les techniques de transformation pour obtenir une version itérative

A noter que ces techniques ne sont pas à 100% automatisables (pour l'instant !). Dans des cas non triviaux, une intervention humaine peut être nécessaire. Exemple avec une récursivité terminale :

Récursivité terminale

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

Version itérative

```
f_iter(X) =  
  si p(X) = faux alors  
    repeter  
      b(X);  
      X := nouveau(X) ;  
  jusque p(X) = vrai;  
  a(X);
```

Réversivité

Avantages et inconvénients de la réversivité

Question :

Quels sont selon vous les avantages et inconvénients de l'approche réversive pour la conception d'algorithmes ?

Avantages et inconvénients de la réversivité

Question :

Quels sont selon vous les avantages et inconvénients de l'approche réversive pour la conception d'algorithmes ?

Avantages

- ▶ Simple, élégant, concis
- ▶ Naturelle pour certains problèmes
- ▶ Similarité avec la preuve par induction

Inconvénients

- ▶ Peut devenir coûteux en espace mémoire si mal écrit
- ▶ Demande un effort pour mise en œuvre
- ▶ Les algorithmes récursifs nécessitent une pile (d'appels).

Complexité

Complexité

Définition

La **complexité d'un algorithme** est l'estimation formelle de la quantité de ressources nécessaire afin d'exécuter un algorithme. Ces ressources peuvent être en temps, espace mémoire, stockage, etc. La quantité peut être évaluée dans le pire, meilleur des cas, ou cas moyen.

La **complexité d'un algorithme** est l'estimation formelle de la quantité de ressources nécessaire afin d'exécuter un algorithme. Ces ressources peuvent être en temps, espace mémoire, stockage, etc. La quantité peut être évaluée dans le pire, meilleur des cas, ou cas moyen.

Il existe différents types de complexité :

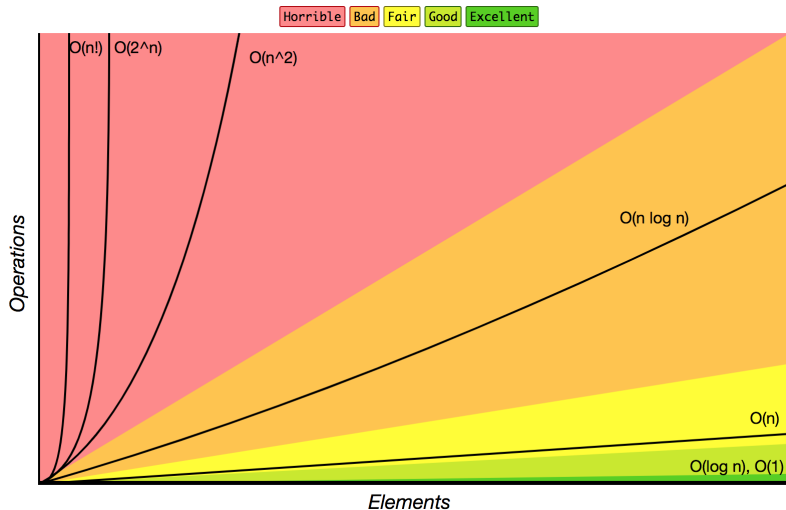
- ▶ **Meilleur des cas** : plus *petit* nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée.
- ▶ **Pire des cas** : c'est le plus *grand* nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée.
- ▶ **Cas moyen** : c'est la moyenne des complexités de l'algorithme sur des jeux de données de taille fixée.

Note : C'est souvent l'analyse du pire des cas qui est choisie (donne une limite supérieure de performance). On s'intéresse le plus souvent à la complexité en nombre d'opérations.

Complexité

Types de complexité

Courbes de croissances de fonctions⁵ de complexité.



⁵<http://bigocheatsheet.com/>

Complexité

Types de complexité

Notation	Complexité	Intuition
$\mathcal{O}(1)$	constante	<i>premier ou n-ème élément d'une liste, ..</i>
$\mathcal{O}(\log n)$	logarithmique	<i>coupe en deux et encore, ..</i>
$\mathcal{O}(n)$	linéaire	<i>parcours données, ..</i>
$\mathcal{O}(n \log n)$	quasi-linéaire	<i>couper en deux et combine, ..</i>
$\mathcal{O}(n^2)$	quadratique	<i>parcours données avec 2 boucles, ..</i>
$\mathcal{O}(2^n)$	exponentiel	<i>on teste toutes les combinaisons, ..</i>
$\mathcal{O}(n^k)(k > 2)$	polynomial	<i>parcours données avec k boucles, ..</i>
$\mathcal{O}(n!)$	factorielle	<i>on teste toutes les chemins (graphe), ..</i>

Note : noter que les constantes n'importent pas dans le calcul de la complexité. Par exemple une complexité de $\mathcal{O}(2n)$ sera équivalente à une complexité $\mathcal{O}(n)$.

Complexité

Calcul de la complexité

```
def maximum(L):  
    m=L[0]  
    for i in range(1,len(L)):  
        if L[i]>m:  
            m=L[i]  
    return m
```

Complexité

Calcul de la complexité

```
def maximum(L):  
    m=L[0]  
    for i in range(1,len(L)):  
        if L[i]>m:  
            m=L[i]  
    return m
```

$\mathcal{O}(n)$

```
def nocc(x,L):  
    n=0  
    for y in L:  
        if x==y:  
            n=n+1  
    return n
```

Complexité

Calcul de la complexité

```
def maximum(L):  
    m=L[0]  
    for i in range(1,len(L)):  
        if L[i]>m:  
            m=L[i]  
    return m
```

$\mathcal{O}(n)$

```
def nocc(x,L):  
    n=0  
    for y in L:  
        if x==y:  
            n=n+1  
    return n
```

$\mathcal{O}(n)$

Complexité

Calcul de la complexité

```
def majoritaire(L):  
    xmaj=L[0]  
    nmaj=nocc(xmaj,L)  
    for i in range(1,len(L)):  
        if nocc(L[i],L)>nmaj:  
            xmaj=L[i]  
            nmaj=nocc(L[i],L)  
    return xmaj
```

Complexité

Calcul de la complexité

```
def majoritaire(L):  
    xmaj=L[0]  
    nmaj=nocc(xmaj,L)  
    for i in range(1,len(L)):  
        if nocc(L[i],L)>nmaj:  
            xmaj=L[i]  
            nmaj=nocc(L[i],L)  
    return xmaj
```

$\mathcal{O}(n^2)$

Complexité

Calcul de la complexité

```
def somcubes(n):  
    s = 0  
    while n>0:  
        s = s+(n%10)**3  
        n = n//10  
    return s  
return y  
  
def egaux_somcubes(N):  
    L = []  
    for n in range(0, N+1):  
        if n==somcubes(n):  
            L.append(n)  
    return L  
  
egaux_somcubes ?
```

Complexité

Calcul de la complexité

```
def somcubes(n):  
    s = 0  
    while n>0:  
        s = s+(n%10)**3  
        n = n//10  
    return s  
return y  
  
def egaux_somcubes(N):  
    L = []  
    for n in range(0, N+1):  
        if n==somcubes(n):  
            L.append(n)  
    return L  
  
egaux_somcubes ?
```

$\mathcal{O}(n \log(n))$

Complexité

Calcul de la complexité

Exercice. Vous disposez de deux listes [1, 3, 8, 10] et [2, 3, 9] déjà triées et vous souhaitez obtenir une nouvelle liste fusion de ces deux listes (sans utiliser les fonctions de tri sort/sorted). Quelle est la complexité ?

Exemple d'exécution :

```
print(fusion([2, 2, 3], [0, 4, 5, 14, 20, 25]))  
# [0, 2, 2, 3, 4, 5, 14, 20, 25]
```

Complexité

Calcul de la complexité

Exercice. Vous disposez de deux listes [1, 3, 8, 10] et [2, 3, 9] déjà triées et vous souhaitez obtenir une nouvelle liste fusion de ces deux listes (sans utiliser les fonctions de tri sort/sorted). Quelle est la complexité ?

Exemple d'exécution :

```
print(fusion([2, 2, 3], [0, 4, 5, 14, 20, 25]))  
# [0, 2, 2, 3, 4, 5, 14, 20, 25]
```

Solution.

- ▶ Similaire à l'algorithme du tri fusion
- ▶ On parcourt toutes les données 1 fois donc $O(N)$

<https://gitlab.ec-lyon.fr/rvuillemin/inf-tc1/-/blob/master/TD01/code/listes-fusion.py>

Complexité

Calcul de la complexité

Grand O	\mathcal{O}
Grand Omega	Ω
Grand Theta	Θ
Petit O	o
Petit Omega	ω

L'ordre de croissance de $T(N) \leq f(N) \rightarrow T(N) = O(f(N)) : \textit{big-Oh}$

L'ordre de croissance de $T(N) \geq f(N) \rightarrow T(N) = \Omega(f(N)) : \textit{Omega}$

L'ordre de croissance de $T(N) \approx f(N) \rightarrow T(N) = \Theta(f(N)) : \textit{Theta}$

L'ordre de croissance de $T(N) < f(N) \rightarrow T(N) = o(f(N)) : \textit{little-Oh}$

L'ordre de croissance de ω est $T(N) > f(N)$

https://fr.wikipedia.org/wiki/Comparaison_asymptotique

Complexité

Exemples de complexité

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
<u>Quicksort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
<u>Mergesort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Timsort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Heapsort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
<u>Bubble Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Insertion Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Selection Sort</u>	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Tree Sort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
<u>Shell Sort</u>	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
<u>Bucket Sort</u>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
<u>Radix Sort</u>	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
<u>Counting Sort</u>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
<u>Cubesort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$

<http://bigocheatsheet.com/>

Complexité

Exemples de complexité

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Stack	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Queue	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Singly-Linked List	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Doubly-Linked List	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Skip List	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n \log(n))$
Hash Table	N/A	$\theta(1)$	$\theta(1)$	$\theta(1)$	N/A	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Binary Search Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Cartesian Tree	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	N/A	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
B-Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
Red-Black Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
Splay Tree	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
AVL Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
KD Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$

<http://bigocheatsheet.com/>

Complexité

Calcul de la complexité

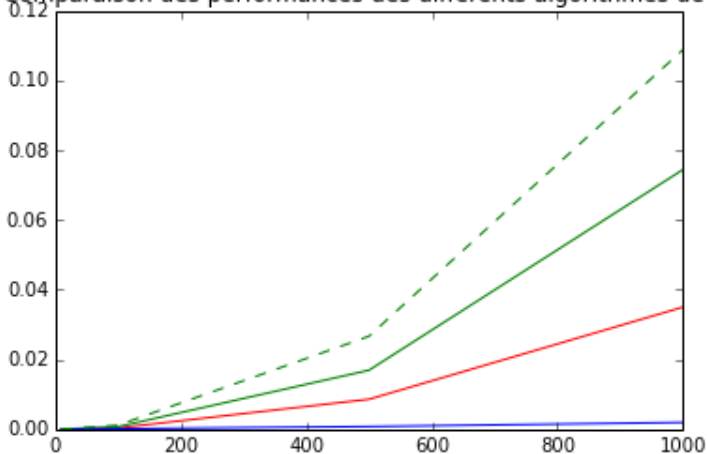
Il n'y a pas qu'une mais plusieurs méthodes pour calculer la complexité d'un algorithme, tout dépend de ses propriétés (et de la précision de la complexité souhaitée). Voici les principales approches :

- ▶ **Réduction du code à un cas connu**, et combinaison des complexités. Par exemple 2 boucles ($O(\log N)$) donnent une complexité globale de $O(N^2 \log(N))$
- ▶ **Réduction à une famille de fonctions connues** et calcul du taux relatif de croissance (limite)
- ▶ **Calcul empirique en affichant les temps d'exécution** en fonction de la taille d'un problème. Pour rappel cela est indépendant de la puissance de la machine.

Complexité Calcul (empirique) de la complexité

Une approche empirique (appelée analyse amortie) consiste à enregistrer les temps d'exécution (axe X) de plusieurs méthodes en fonction de la taille des données (axe Y).

Comparaison des performances des différents algorithmes de tri



Complexité

Calcul (empirique) de la complexité

Un algorithme simple pour calculer le temps d'exécution de fonctions (ex. tris) selon un jeu de données qui varie en taille et en valeurs aléatoires.

```
import time
import random
import matplotlib.pyplot as plt
%%matplotlib inline

nvalues = [100, 500, 1000, 1500, 2000, 2500, 3000]
savedtimes = []

for i in nvalues:

    random.seed()
    liste = []
    for x in range(i): liste.append(random.randint(0, p))

    a=time.perf_counter()
    for n in liste:
        # Pour chaque valeur de la liste en cours
        b=time.perf_counter()
        savedtimes.append(b-a)

    # Répéter pour comparaison 2..

plt.plot(nvalues, savedtimes1, "r-", label="Comparaison 1")
plt.plot(nvalues, savedtimes2, "g-", label="Comparaison 2")
plt.title("Comparaison des performances")
```

Sommaire

Introduction

- Exemple d'algorithme
- Objectifs du module INF-TC1
- Ressources

Principes d'algorithmique

- Définitions
- Schémas d'algorithmes
- Exemples d'algorithmes

Principes généraux

- Récursivité
- Complexité

Types et structures de données

- Tables de Hachage
- Piles et files
- Files de priorité
- Listes liées

Types et structures de données

Types et structures de données de base

Les variables en programmation se voient associer un *type*, afin de déterminer la nature des données qui y sont stockées et les opérations sur elles. Voici les principaux types (et équivalent Python) :

- ▶ Entier (<class 'int'>) : 1, 2, ..
- ▶ Flottant (<class 'float'>) : 1.1, 5.6, ..
- ▶ Caractère (<class 'str'>) : 'A', 'a', ..
- ▶ Chaîne de caractères (<class 'str'>) : "ABC" (manipulables comme des tableaux)

Il est possible de convertir les variables en autres types : `int()`, `float()`, `str()`, .. Garder en tête : toutes les variables Python sont des objets et possèdent des méthodes⁶

A noter que les types sont importants dans certains langages de programmation (Python facilite grandement le typage)⁷

⁶<https://docs.python.org/3/library/functions.html>

⁷<https://sites.uclouvain.be/SystInfo/notes/Theorie/html/C/datatypes.html>

Types et structures de données de base

Au delà des types, il existe des structures de données souvent appelées *structures de données implicites* car sont très efficace (ex: accès en temps constant $O(1)$ pour une liste et ne demandent pas beaucoup de ressources)

Exemples des principales structures de données (en Python) :

- ▶ Liste (<class 'list'>) : [], [1, 2], ..
- ▶ Ensemble (Set) : 'apple', 'orange', 'apple', 'pear'
- ▶ Tuple (<class 'tuple'>) : (1, 2), ..
- ▶ Dictionnaire (Dict) :

Types et structures de données

Types et structures de données de base

Comment choisir un type ou une structure de données ?

Types et structures de données de base

Comment choisir un type ou une structure de données ?

- ▶ En fonction des besoins de l'algorithme
- ▶ En fonction des ressources mises à disposition et de la complexité de l'algorithme (espace disque, mémoire, temps ..)
 - ▶ Par exemple un arbre binaire pour la recherche $O(\log(N))$
 - ▶ ..mais l'insertion est très coûteuse !

A noter :

- ▶ Dans la pratique il est souvent nécessaire de les étendre pour résoudre des problèmes plus complexes
- ▶ Par exemple les arbres et graphes possèdent des listes de nœuds et de voisins.

Types et structures de données

Tables de Hachage (Hash Table)

Définition. Les **tables de hachage** sont des tableaux où chaque point d'entrée est associé à une unique valeur, selon une fonction d'association. Cela permet un accès en temps constant aux données.

```
>>> phonebook = {'bob': 7387, 'alice': 3719, 'jack': 7052}
>>> phonebook['alice']
3719
```

- ▶ Implémentation en dictionnaire Python
- ▶ L'assertion `KeyError: 'missing'` est levée si accès à une clé non-définie
- ▶ Une bonne pratique est d'utiliser `.get("attr", "")` afin de renvoyer une valeur par défaut si la clé n'existe pas
- ▶ Nous verrons qu'elles sont très utilisées pour la mémorisation afin d'éviter de re-faire certains calculs (ex. prog. dynamique)

Types et structures de données

Tables de Hachage (Hash Table)

On souhaite parfois conserver l'ordre d'ajout des éléments il faut pour cela utiliser le module collections.

```
>>> import collections
>>> d = collections.OrderedDict(one=1, two=2, three=3)
# OrderedDict([('one', 1), ('two', 2), ('three', 3)])

>>> d['four'] = 4
# OrderedDict([('one', 1), ('two', 2), ('three', 3),
# ('four', 4)])

>>> d.keys()
odict_keys(['one', 'two', 'three', 'four'])
```

Utilisation de valeurs par défaut :

```
d = defaultdict(lambda : 6)
d["k"] += 1
print(d["k"]) # 7
```

Exemple d'utilisation de table de hachage

Afficher tous les entiers tels que $A^2 + B^2 = C^2 + D^2$ avec A, B, C, D variant de 1 à 1000.

```
n = 1000
for a from 1 to n
  for b from 1 to n
    for c from 1 to n
      for d from 1 to n
        if (a^2 + b^2 == c^2 + d^2)
          print (a, b, c, d)
```

Exemple d'utilisation de table de hachage

Afficher tous les entiers tels que $A^2 + B^2 = C^2 + D^2$ avec A, B, C, D variant de 1 à 1000.

```
n = 1000
for a from 1 to n
  for b from 1 to n
    for c from 1 to n
      for d from 1 to n
        if (a^2 + b^2 == c^2 + d^2)
          print (a, b, c, d)
```

```
n = 1000
for c from 1 to n
  for d from 1 to n
    result = c^3 + d^3
    append(c,d) to list at value map[result]
```

```
for a from 1 to n
  for b from 1 to n
    result = a^3 + b^3
    list = map.get(result)
    for each pair in list
      print(a, b, pair)
```

Types et structures de données

Ensembles (Sets)

Définition. Les **ensembles** permettent de stocker plusieurs éléments de manière unique, non-ordonnés. Ils permettent des opérations booléennes.

```
>>> set_int = {1, 2, 3}
>>> print(set_int) # {1, 2, 3, 4}

>>> set_mixe = {1.0, "Hello", (1, 2, 3)}
>>> print(set_mixe) # {1.0, (1, 2, 3), 'Hello'}
```

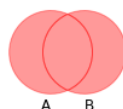
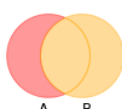
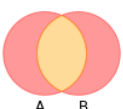
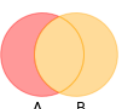
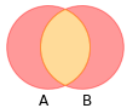
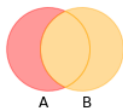
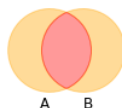
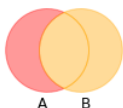
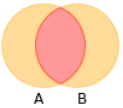
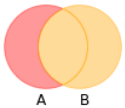
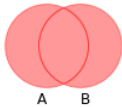
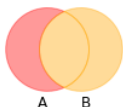
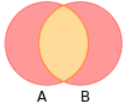
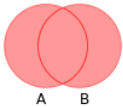
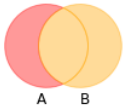
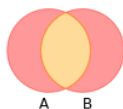
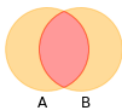
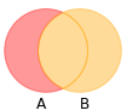
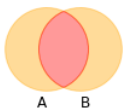
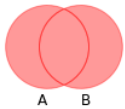
>>> # attention car pour initialiser un set vide
>>> # ne pas utiliser set_empty = {} # dict
>>> set_empty = set()

>>> # on ne peut pas mettre d'objets mutables
>>> set_liste = {1, 2, [3, 4]}

```
Traceback (most recent call last):
  File "<string>", line 15, in <module>
    my_set = {1, 2, [3, 4]}
TypeError: unhashable type: 'list'
```

Types et structures de données

Ensembles (Sets)



Types et structures de données

Ensembles (Sets)

<code>add()</code>	ajout d'un élément
<code>clear()</code>	supprime tous les éléments
<code>copy()</code>	renvoie la copie d'un ensemble
<code>difference</code>	la différence de deux ensembles
<code>intersection()</code>	intersection de deux ensembles
<code>pop()</code>	un élément aléatoire
<code>union()</code>	union de deux ensembles
<code>isdisjoint()</code>	True si disjoint
<code>issubset()</code>	True si sous-semble
<code>issuperset()</code>	True si est contenu par l'ensemble

- ▶ Il existe de nombreuses autres opérations sur les ensembles
- ▶ Les `frozenset` permettent de rendre l'ensemble immuable

<https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset>

<https://docs.python.org/3/tutorial/datastructures.html>

Définition. Les **tuples** sont un ensemble ordonné d'éléments sous forme de n-uplets. Ils ne peuvent pas être modifiés une fois créés (contrairement aux listes).

```
>>> a = (3, 4, 7)
>>> type(a)
>>> <class 'tuple'>

>>> (b, c) = (5, 6)

>>> tuple_vide = tuple()
>>> tuple_vide = ()

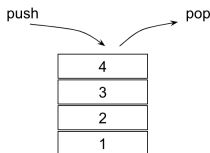
>>> tuple_1_element = (5, )

>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped) # [(1, 4), (2, 5), (3, 6)]
```

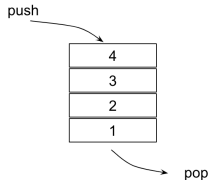
- La fonction `zip` permet d'aggréger des tuples

Définition. Les **piles** et les **files** permettent de manipuler des valeurs (ou objets) de manière séquentielle. Elles ont deux principales opérations : ajout (*push*) et enlève (*pop*), mais avec des stratégies d'ordre différentes :

Pile



File



- ▶ L'élément inséré en premier est en tête
- ▶ Facilement réalisable avec une simple liste !
- ▶ La classe Queue propose en fait plusieurs structures de données (avec des application en programmation concurrente).
- ▶ A noter que les piles et files définissent les opérations et leurs résultats, mais **pas leur implémentation**.

Types et structures de données

Piles et Files

Code. L'implémentation des piles et des files en utilisant les List.

Piles (LIFO)

```
stack = [3, 4, 5]
stack.append(6)
stack.append(7)
print(stack)
>>> [3, 4, 5, 6, 7]
stack.pop()
>>> 7
print(stack)
>>> [3, 4, 5, 6]
stack.pop()
>>> 6
stack.pop()
>>> 5
print(stack)
>>> [3, 4]
```

Files (FIFO)

```
queue = [3, 4, 5]
queue.append(6)
queue.append(7)
print(queue)
>>> [3, 4, 5, 6, 7]
queue.pop(0)
>>> 3
print(queue)
>>> [4, 5, 6, 7]
queue.pop(0)
>>> 4
queue.pop(0)
>>> 5
print(queue)
>>> [6, 7]
```

Types et structures de données

Piles et Files

- ▶ La différence entre piles et file réside dans la méthode d'accès aux données : pour les Piles (Last In First Out - LIFO) et les Files (First In First Out (FIFO))
- ▶ La classe Queue propose en fait plusieurs structures de données (avec des application en programmation concurrente).

File

```
import queue

file = queue.Queue()

for i in range(5): file.put(i)

while not file.empty():
    print(file.get(), end=" ")
```

```
>>> 0 1 2 3 4
```

Pile

```
import queue

pile = queue.LifoQueue()

for i in range(5): pile.put(i)

while not pile.empty():
    print(pile.get(), end=" ")
```

```
>>> 4 3 2 1 0
```

Types et structures de données

Complexité des piles et files

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Stack	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Queue	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Singly-Linked List	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Doubly-Linked List	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Skip List	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n \log(n))$
Hash Table	N/A	$\theta(1)$	$\theta(1)$	$\theta(1)$	N/A	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Binary Search Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Cartesian Tree	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	N/A	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
B-Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
Red-Black Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
Splay Tree	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
AVL Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
KD Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$

<http://bigocheatsheet.com/>

Éléments communs aux piles et files

<code>empty()</code>	test de vacuité
<code>full()</code>	plein (si on a donné une taille max à la création)
<code>get()</code>	retourne (et supprime un élément)
<code>put()</code>	ajoute un élément
<code>qsize()</code>	retourne la taille de la liste
<code>reverse()</code>	inverse l'ordre des éléments

- ▶ A noter que les structures de Pile et File en Python utilisable dans les Threads (structure *Thread-Safe*).
- ▶ Une variation des files est la *file de priorité*

<https://docs.python.org/3/tutorial/datastructures.html>

Files de priorité

Définition. Une **file de priorité** est une file (ou pile ou liste) qui renvoie un élément basé sur les caractéristiques d'une variable (priorité).

Définition. Une **file de priorité** est une file (ou pile ou liste) qui renvoie un élément basé sur les caractéristiques d'une variable (priorité).

Propriétés.

- ▶ Pour une variable quantitative le minimum ou maximum de la file. Pour d'autre type de variable (ex. catégories) toute relation d'ordre est valable.
- ▶ Des files peuvent avoir le même comportement mais un état interne différent : soit constamment mis à jour, soit après les lecteurs/écritures.
- ▶ L'état interne peut juste être préservé avec une fonction de tri et donc optimiser la complexité de la structure de données

Types et structures de données

File de priorité

Code. Implémentation avec tri lors de la *suppression*.

```
class PriorityQueue(object):  
    def __init__(self):  
        self.queue = []  
  
    # Pour tester si vide  
    def isEmpty(self):  
        return len(self.queue) == 0  
  
    # Pour ajouter  
    def insert(self, data):  
        self.queue.append(data)
```

Pour supprimer

```
def delete(self):  
    try:  
        max = 0  
        for i in range(len(self.queue)):  
            if self.queue[i] > self.queue[max]:  
                max = i  
        item = self.queue[max]  
        del self.queue[max]  
        return item  
    except IndexError:  
        print()  
        exit()
```

Pour afficher

```
def __str__(self):  
    return ' '.join([str(i) for i in self.queue])
```

Types et structures de données

File de priorité

```
if __name__ == '__main__':  
    myQueue = PriorityQueue()  
    myQueue.insert(12)  
    myQueue.insert(1)  
    myQueue.insert(14)  
    myQueue.insert(7)  
    print(myQueue)  
    while not myQueue.isEmpty():  
        print(myQueue.delete())
```

12 1 14 7

14

12

7

1

()

Définition. Un **Tas** permet de stocker des données tout en conservant une de leur propriété, par exemple une relation d'ordre.

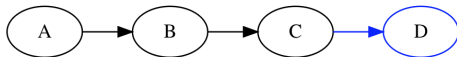
Exemple d'implémentation avec le module `heapq` de Python.

```
>>> from heapq import heapify, heappush, heappop
>>> heap = [10, 8, 1, 2, 4, 9, 3, 4, 7]
>>> heapify(heap)
>>> heap
[1, 2, 3, 4, 4, 9, 10, 8, 7]
>>> heappop(heap)
1
>>> heap
[2, 4, 3, 4, 7, 9, 10, 8]
>>> heappush(heap, 5)
>>> heap
[2, 4, 3, 4, 7, 9, 10, 8, 5]
```

Types et structures de données

Listes liées (*Linked lists*)

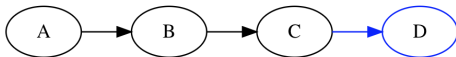
Définition. Une **liste liée** est une séquence de valeurs (ou objets) appelés *nœuds* qui sont reliés entre eux afin de permettre leur stockage et recherche.



Types et structures de données

Listes liées (*Linked lists*)

Définition. Une **liste liée** est une séquence de valeurs (ou objets) appelés *nœuds* qui sont reliés entre eux afin de permettre leur stockage et recherche.



Propriétés.

- ▶ Le premier nœud s'appelle la tête (head), le dernier nœud la queue (tail) et pointe vers `null`
- ▶ Cette structure permet une approche flexible pour manipuler les objets : augmenter leur nombre, ordre, etc.
- ▶ Surtout permet d'**allouer dynamiquement la mémoire**, là où un tableau a besoin d'allouer toute la place avant d'être rempli.
- ▶ Par contre nécessite un temps de recherche linéaire (contrairement aux tableaux), et peut poser un soucis pour implémenter une pile.

Types et structures de données

Listes liées

Code. Un objet Node pour chaque noeud de la liste (et son voisin, si voisin il y a); et ensuite un objet LinkedList pour identifier la tête de la liste et les méthodes de parcours.

```
class Node:
    def __init__(self, data = None, next = None):
        self.data = data
        self.next = next

class LinkedList:
    def __init__(self):
        self.head = None

    def listprint(self):
        printval = self.head
        while printval is not None:
            print(printval.dataval)
            printval = printval.nextval
```


Types et structures de données

Listes liées

Exemple d'utilisation d'une liste liée

```
list = LinkedList()
list.head = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")
list.head.next = e2
e2.next = e3
list.listprint()
```

```
>>> Mon
```

```
>>> Tue
```

```
>>> Wed
```

Types et structures de données

Listes liées

Code. Implémentation d'un itérateur permet de parcourir une classe qui contient elle-même un élément itérable.

Exemples d'utilisation :

```
def __iter__(self):  
    node = self.head  
    while node is not None:  
        yield node  
        node = node.next
```

Exemple

```
>>> llist = LinkedList(["a", "b", "c", "d", "e"])  
>>> llist  
a -> b -> c -> d -> e -> None  
  
>>> for node in llist:  
...     print(node)  
a  
b  
c  
d  
e
```

Exercice. Vérifier qu'une liste chaînée ne contient pas de boucle (autrement dit qu'on ne repasse pas indéfiniment par le même noeud).

Types et structures de données

Listes liées

Exercice. Vérifier qu'une liste chaînée ne contient pas de boucle (autrement dit qu'on ne repasse pas indéfiniment par le même noeud).

```
from LinkedList import *

ll = LinkedList() # Ajout de données
ll.push(20)
ll.push(4)
ll.push(15)
ll.push(10)

# Création d'une boucle
ll.head.next.next.next.next = ll.head;

if( ll.detectLoop()):
    print ("Il y a une boucle !")
else :
    print ("Pas de boucle ! ")
```

Types et structures de données

Listes liées

```
def push(self, n):
    if self.head is None:
        self.head = Node(n)
    else:
        node = self.head
        while node.next is not None:
            node = node.next
        node.next = Node(n)

def detectLoop(self):
    s = set()
    temp = self.head
    while (temp):
        if (temp in s):
            return True
        s.add(temp)
        temp = temp.next
    return False
```

Types et structures de données

Listes liées

Question :

Quelle est la complexité de la manipulation de listes liées (non-ordonnées) ? En fonction de l'algorithme (recherche, insertion, suppression) dans les cas moyen et pire des cas.

Algorithm	Average	Worst Case
-----------	---------	------------

Types et structures de données

Listes liées

Question :

Quelle est la complexité de la manipulation de listes liées (non-ordonnées) ? En fonction de l'algorithme (recherche, insertion, suppression) dans les cas moyen et pire des cas.

Algorithm	Average	Worst Case
-----------	---------	------------

Search	$O(n)$	$O(n)$
--------	--------	--------

Insert	$O(1)$	$O(1)$
--------	--------	--------

Delete	$O(1)$	$O(1)$
--------	--------	--------

Structures de données linéaires vs non-linéaires. Nous n'avons vu que des structures de données principalement *linéaires*, qui ont l'avantage d'être simples et implémentées en Python. Dans le prochain cours nous verrons les structures *non-linéaires* plus efficaces mais aussi plus compliquées à utiliser et implémenter.

Linéaires

- ▶ Ex : tableaux, piles, files
- ▶ Stockage séquentiel en 1 dimension
- ▶ Utilisation de la mémoire pas toujours efficace
- ▶ Augmentation du temps de parcours avec la taille du jeu de données

Non-linéaires

- ▶ Ex : arbres, graphes
- ▶ Organisation hiérarchique des données
- ▶ Plusieurs parcours sont parfois nécessaires
- ▶ Le temps de parcours peut être optimisé

A noter qu'il existe des structures de données hybrides, telles que les *files de priorité* stockées sous forme de tableau mais parcourues comme des arbres.