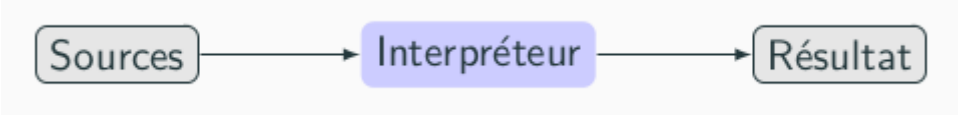
Algo1 : Synthèse Cours

Cours 1 :

- Langage machine (=code machine, code natif ou code binaire) : directement compréhensible par la machine ; code purement numérique opérations très basiques et/ou des données
- Langage d'assemblage : langage symbolique textuel d'un peu plus haut niveau, traduisible en langage machine par un assembleur
- Langage de haut niveau : doit être compilé ou interprété pour être compris par la machine
- Langage de programmation : langage informatique permettant à un humain d'écrire un code source qui sera analysé par un ordi
- Paradigmes de programmation :
  - Programmation impérative : l'exécution d'une séquence d'instructions va modifier l'état des données (la plupart des langages sont impératifs)
  - Programmation objet : à base de classes, façon d'organiser et de regrouper les données et les traitements qui y sont liés
  - Programmation fonctionnelle : déclarative, qui se rapproche des théories mathématiques
- 3 modes d'execution des langages de prog :
  - langages compilés :
    - Exécutée par un programme : compilateur
    - Traduction du code source en une fois en code machine
    - Le code machine est directement exécutable par la machine, indépendamment du compilateur
    - Peut être conservé tel quel dans un fichier (fichier exécutable)
    - Exemples : Pascal, Ada, C, C++, Fortran...

  - langages interprétés :
    - Exécutée par un programme : interpréteur
    - Chaque ligne du code source est traduite au fur et à mesure en instructions directement exécutées
    - Pas de génération de programme objet
    - Technique très souple, mais peu performante : l'interprétation doit être réalisée à chaque exécution
    - Exemples : LISP, Basic, Perl, PROLOG...



- langage semi-compilés :
  - Combinaison des deux techniques précédentes
  - Le compilateur produit un code intermédiaire appelé bytecode
  - L'interpréteur l'exécute pour produire le résultat
  - Le bytecode est facile à interpréter en langage machine
  - Exemples : Java, Python...

- Python :
  - Crée en 1989 par Guido van Rossum
  - Langage semi-compilé orienté objet
  - 2 modes d'exécution :
    - mode interprété (=calculatrice python) : permet l'utilisation interactive de l'interpréteur
    - utilisation de scripts Python :
      - l'utilisation de l'interpréteur présente vite des limites dès qu'on veut exécuter une suite d'instructions plus complexe
      - enregistrer ces instructions dans un fichier d'exentension standard : .py
- Une variable :
  - C'est une zone de la mémoire dans laquelle une valeur est stockée
  - Elle possède 4 propriétés : nom (programmeur), adresse(ordinateur), type, valeur
  - En python, la déclaration d'une variable et son initialisation se font en même temps
  - Les noms des variables (et des fonctions) sont appelés des identificateurs
- Python 3 contient 33 mots clés :

```
and       del       from      None      True
as        elif      global    nonlocal  try
assert    else      if        not       while
break     except    import    or        with
class     False     in        pass      yield
continue  finally   is        raise
def       for       lambda    return
```

- Il faut veiller à ce que les identificateurs soient aussi explicite que possible
- Importance d'utiliser une politique cohérente de nommage des identificateurs, tout en maj ou sinon tout en minuscules avec possiblement des majs pour augmenter la lisibilité

- L’affectation : =
- Une comparaison : ==
- type() fournit le type de l’argument, les principaux types de données sont :
  - les entiers : int
  - les réels (ou flottants) : float(avec un point décimal et JAMAIS une virgule)
  - les booléens : bool
  - les chaînes de caractères : str
  - les listes : list
  - les tuples (ou n-uplets) : tuple
  - les dictionnaires : dict
  - les ensembles : set
- Les opérations :
  - l’opposé et la soustraction : -
  - l’addition : +
  - la multiplication : *
  - la puissance : **
  - la division : /
  - la division entière : //
  - le reste de la division entière : %
- Séquences d’échappement :
  - saut de ligne avec ... (placéc en fin de ligne) : \
  - antislash: \\ (même si un ‘’\’’ suffit)
  - apostrophe : \’
  - guillemet : \’’ (touche 3)
  - saut de ligne : \n
  - saut de page : \f
  - retour à la ligne : \r
  - tab horizontale : \t
  - tab verticale : \v

Cours 2 :

- L’indentation est obligatoire et primordiale
- 4 espaces plutôt qu’un tab
- Les opérateurs de comparaison
  - i == j : égalité
  - i != j : différence
  - i < j : strictement inférieur
  - i <= j : inférieur ou égal
  - i > j : strictement supérieur
  - i >= j : supérieur ou égal
- Les opérateurs booléens de base :
  - La négation not (=non)
  - La conjonction and (=et)
  - La disjonction or (=ou)

- Python est paresseux : il n'analysera pas l'intérieur d'une fonction si la condition n'est pas respectée

- Les lois de Morgan :
  - not (a and b) == not(a) or not(b)
  - not (a or b) == not(a) and not(b)
- Dans les instructions conditionnelles, on nomme :
  - la condition : l'expression qui permet de choisir
  - le conséquent : l'instruction à appliquer si le choix est vrai
  - l'alternant : l'instruction à appliquer si le choix est faux
- ATTENTION : la condition est une expression booléenne. On n'écrit jamais : condition == True/False
- if condition1 :
  elif condition2 :
  else :
  ...

Cours 3 :

- while condition :
  instruction_1
  instruction_2
  ...
  instruction_n
- Une boucle while se termine qd sa condition est fausse
- If faut obligatoirement qu'une des instructions du corps de la boucle modifie potentiellement la valeur de la condition de sortie de la boucle, exemple :
  while i condition :
  instruction
  i=i+/-*...
- while se répète tant que la condition est vraie
- Lorsque l'on connait l'ensemble des valeurs à considérer, il peut être pratique de parcourir une séquence de valeurs : la boucle for
- for ... in liste :
  instruction_1
  ...
  - le for s'arrête lorsque la liste a été entièrement parcouru
- range construit un intervalle d'entiers, plusieurs utilisations sont possibles :
  - range(n) : génère les entiers de 0 à n-1
  - range(i, j) : génère les entiers de i à j-1 (si i>j, aucun nombre ne sera généré)
  - range (i, j, k) : génère les entiers i à j-1 séparés par un pas de k
- **Elément de cours** : utiliser `sep` et `end` dans la fonction `print`

  Il est possible de modifier le **séparateur** et la **fin de ligne** dans la commande `print`.
  Quelques exemples :
  — `print(3, 5)` affiche `3 5`
  — `print(3, 5, sep="+++")` affiche `3+++5`
  — `print(3, 5, sep="-", end="@")` affiche `3-5@` sans retour à la ligne ensuite

Cours 4 :

- Ne confondre la chaîne vide avec la chaîne contenant un espace
- len(‘’... ‘’) /ou/ len(chaine) : retourne la longueur d’une chaîne de caractères, c’est-à-dire le nbre de caractères qui la compose
- l’opératuer + permet de concaténer (=addition l’une à la suite de l’autre) les chaînes de caractères
- chaine = input(‘’...’’)
- n=int(input(‘’...’’))
- chaine[i] est le ième caractère i ds la chaine chaine, l’indice commence à 0
- chaine[-i] est le ième caractère -i ds la chaine chaine, l’indice inverse commence à -1
- chaine[i : j] permet d’accéder à une portion de la chaîne (=sous-chaîne)
- les chaînes peuvent être comparées avec des == et !=
- str(val) : converti en str la variable val
- s.lower() : retourne la chaine s où les caractères ont été mis en minuscule
- s.upper() : retourne la chaine s où les caractères ont été mis en majuscule
- s.capitalize() : retourne la chaine s où la première lettre du premier mot est en majuscule, les autres en minuscule
- s.title() : retourne la chaine s où la première lettre de chaque mot est en majuscule, les autres en minuscule
- s.swapcase() : retourne la chaine s où les lettres majuscules et minuscules sont inversées
- une liste :
  - de type list
  - collection ordonnée et modifiable d’éléments éventuellemtn hétérogènes
  - formée d’éléments séparés par des virgules et délimitée par des [...]
  - liste vide [] :
    - liste contenant aucun élément
    - c’est aussi l’élément neutre
- len(liste) donne le nbre d’éléments d’une liste
- on peut comparer des listes avec == et !=
- l’opérateur + permet de concaténer les listes
- les types que nous avons traités jusque ici (int, bool, str) sont immutables
- un objet mutable peut être modifié : remplacement, suppression ou ajout d’une partie de l’objet, les listes sont mutables
- liste.append(...) permet d’ajouter un élément à la fin de la liste depuis laquelle elle est appelée
- l’indice d’un élément dans une liste est sa position dans la liste
- liste[i] est le ième élément i ds la chaine chaine, l’indice commence à 0
- liste[-i] est le ième élément -i ds la chaine chaine, l’indice inverse commence à -1
- Découpage avec pas positif :
  - liste [i:j:k] permet d’accéder à tous les éléments dans le liste, compris entre les indices i (inclus) et j (exclus) avec un pas de k

- Découpage avec pas négatif :
  - liste [i:j:-k] permet d'accéder à tous les éléments dans le liste, compris entre les indices i (inclus) et j (exclus) avec un pas de k dans l'indice inverse
- un set {}:
  - type set
  - collection non ordonnée d'éléments uniques
  - ensemble est formé d'éléments séparés par des virgules, et entourés d'accolades
  - ensemble vide, noté set(), est un ensemble qui ne contient aucun élément
  - un set est une transposition informatique de la notion d'ensemble mathématiques
- Soit E et F deux ensembles, et x un élément quelconque (notation maths) :
  - len(E) : le cardinal de E (|E|)
  - set() : l'ensemble vide (∅)
  - x in E : l'appartenance (∈)
  - x not in E : la non-appartenance (∈barré)
  - E < F : l'inclusion stricte (E ⊂ F)
  - E <= F : l'inclusion large (E ⊆ F)
  - E & F : l'intersection (E∩F)
  - E | F : l'union (E∪F)
  - E - F : la différence (E\F)
- Les ensembles sont mutables
- Les ensembles ne sont pas ordonnés dont la notion d'indice n'a pas de sens
- Quelques formules, soit s un ensemble :
  - s.add() : ajoute un élément à s
  - s.update() : ajoute plusieurs éléments à s
  - s.remove() : supprime un élément de s s'il appartient à l'ensemble sinon il retourne une erreur
  - s.discard() : supprime un élément de s, s'il n'appartient pas à l'ensemble, il n'y a pas d'erreurs
  - s.clear : supprime tous les éléments de s

Cours 5 :

- Structuration des programmes : les fonctions et les procédures permettent de décomposer un programme complexe en une série de sous-programmes plus simple
- Fonction : bloc d'instructions nommé et paramétré, réalisant une tâche donnée.
  - Elle admet n paramètres et retourne toujours un résultat
- Procédure : bloc d'instructions nommé et paramétré, réalisant une tâche donnée.
  - Elle admet n paramètres et ne retourne pas de résultats

- Une fonction vaut qqch alors qu'une procédure fait qqch

- def nom_fonction(paramètre1, etc) :
  '' '' '' chaîne de documentation '' '' ''
  bloc_d'instructions
- Le nom de la fonction doit respecter les règles suivantes :
  - Aucun caractère spécial (hormis '' _ ''), aucun caractères accentué
  - Commence par une minuscule (les majs sont utilisés pour les classes)
    - Choisir un nom suffisamment explicite
  - la liste de paramètres peut être vide
  - la chaîne de documentations : facultative mais fortement conseillée. Elle est traitée comme un simple commentaire par Python mais elle est mémorisée à part dans un système de documentation interne automatique. Elle doit contenir :
    - la signature de la fonction (type de l'entrée → type de la sortie)
    - une liste d'expression booléennes qui précises les conditions d'application de la fonction si besoin
    - une phrase qui explique ce que fait la fonction
  - pour appeler la fonction : nom_fonction(…)
- Jeu de test (assert) : Les fonctions doivent être testées grâce à l'instruction assert
  - Les jeux de test servent au programmeur pour valider la définition d'une fonction
  - Ils doivent couvrir tous les cas possibles
    - les cas de base
    - les cas extrêmes
  - Ils n'affichent rien, sauf si un test ne passe pas
- Les variables définies à l'intérieur du corps d'une fonction ou d'une procédure ne sont accessibles qu'à la fonction elle-même. Ce sont des variables locales à la fonction.
  - Le contenu des variables locales est inaccessible depuis l'extérieur de la fonction
  - Les variables définies à l'extérieur d'une fonction ou procédure sont des variables locales. Leur contenu est visible de l'intérieur d'une fonction, mais la fonction ne la modifie pas

Cours 6 :

- Module (= module de fonctions) : fichier qui regroupe des ensemble de fonctions, ou des bibliothèques comme des classes ou données par exemple. Ils permettent de :
  - ré-utiliser un code
  - isoler des fonctionnalités particulières dans un espace identifié

- Un module est un fichier en .py écrit en python contenant des définitions et des instructions qui peut être destiné :
  - à être directement exécuté lorsqu'il est court et effectue une action ré-utilisable, on parle souvent d'un script
  - à être utilisé par un autre module. Il exporte alors un certain nombre de fonctionnalités
- Pour importer un module : import nom_fichier
  nom_fichier.fonction
  from nom_fichier import fonction

- Le type matrice n'existe pas, d'où l'utilisation d'int et de list
  - 2 méthodes :
    - en utilisant une liste :

```
#En utilisant une liste
def matrice(a,b,c,d) :
    """Int x Int x Int x Int --> List
    Retourne une matrice 2x2 formée des 4 entiers en entrée"""
    return [a, b, c, d]

def matrice_ref(M, li, co):
    """List x Int x Int --> Int. Retourne l'entier M(li, co)"""
    return M[2 * li + co]
```

    - en utilisant une liste de listes :

```
#En utilisant une liste de listes
def matrice(a,b,c,d) :
    """Int x Int x Int x Int --> List
    Retourne une matrice 2x2 formée des 4 entiers en entrée"""
    return [[a, b], [c, d]]

def matrice_ref(M, li, co):
    """List x Int x Int --> Int. Retourne l'entier M(li, co)"""
    return M[li][co]
```

- Liste complète des modules de base en Python : https://docs.python.org/fr/3/py-modindex.html , dont voici qq exemples :
  - math : fonctions et constantes mathématiques de base (sin, vos, exp, pi,etc)
  - cmath : fonctions et constantes mathématiques avec des nbres complexes
  - sys : interaction avec l'interpréteur Python, passage d'arguments
  - random : génération de nbre aléatoires
  - fractions : fournit un support de l'arithmétique des nombres rationnels
  - turtle : permet de réaliser très simplement des dessins géométriques
    - turtle.setup (width, height)
    - turtle.speed(''...'') : slowest, slow, normal, fast, fastest

- nom=turtle.Turtle()
- turtle.penup()
- turtle.pendown()
- turtle.goto(x,y)
- turtle.left(angle)
- turtle.right(angle)
- turtle.forward(nbre)
- turtle.backward(nbre)
- turtle.circle(+-nbre, angle)
- turtle.hideturtle()
- turtle.exitonclick()

Cours 7 :

- Un algorithme est une suite ordonnée d'instructions qui indique la démarche à suivre pour résoudre une série de problèmes équivalents. Un algorithme (ou un programme) a plusieurs aptitudes :
  - Validité : aptitude à réaliser exactement la tâche pour laquelle il a été conçu
  - Robustesse : aptitude à se protéger des conditions anormales d'utilisation
  - Réutilisabilité : aptitude à être réutilisé pour résoudre des tâches équivalentes à celle pour laquelle il a été conçu
  - Efficacité : aptitude à utiliser de manière optimale les ressources matériel qui l'exécute
- Un programme est une suite d'instructions définies dans un langage donné. Un programme permet de décrire un algorithme.
  - Un algo exprime la structure logique d'un prog : il est indépendant du langage de programmation
  - La traduction de l'algo dans un langage de programmation dépend du langage choisi
- Pourquoi étudier la complexité des algorithmes ?
  - Pour savoir si un algo est efficace ou non
  - Pour savoir comparer deux algo accomplissant la même tâche
  - Pour chaque algo, on veut déterminer indépendamment de l'implémentation (langage/machine choisi pour programmer) :
    - le temps d'exécution
    - la place utilisée en mémoire
  - On ne veut pas : « l'algo A, implémenté sur la machine M ds le langage L et exécuté sur la donnée D utilise k secondes de calcul et j bits de mémoire »
  - On veut : « Quels que soient l'ordi et le langage utilisés, l'algo A_1 est meilleur que l'algo A_2, pour des données de grandes tailles »
- Qu'est ce que la complexité d'un algo ?
  - Il s'agit de caractériser le comportement d'un algo sur l'ensemble D_n des données de taille n
  - La complexité dépend en général de la taille n des données

    - Plusieurs types de complexité :
        - En temps
        - En espace
- Opérations significatives : le temps d'exécution d'un algo est toujours proportionnel au nombre de ces opérations
    - Si plusieurs opérations significatives différentes sont choisies, elles doivent être décomptées séparément en changeant le nombre d'opérations significatives, on varie le degrés de précision de l'analyse
- $coût_A(d)$: complexité de l'algo A sur la donnée $d \in D_n$ de taille n
- Complexité au meilleur des cas :
    - $coût\ min_A(n) = min\{coût_A(d), d \in D_n\}$

- Complexité au pire des cas :
    - $coût\ max_A(n) = max\{coût_A(d), d \in D_n\}$

- Complexité moyenne :
    - $coût\ moy_A(n) = \sum_{d \in D_n} coût_A(d) * p(d)$
- Si deux algorithmes différents effectuent le même travail, il est nécessaire de pouvoir comparer leur complexité, il faut alors connaître la rapidité de croissance des fonctions qui mesurent la complexité lorsque la taille de données croît. On recherche alors l'ordre de grandeur asymptotique, c'est-à-dire le coût de l'algo à la limité lorsque n devient infini
- Plus la taille des données est grande, plus les écartes en temps se creusent
    - Les algo utilisables pour les données de grande taille sont ceux qui s'exécutent en un temps :
        - constant
        - logarithmique (ex : recherche dichotomique)
        - linéaire (ex : recherche séquentielle)
        - nlog(n) (ex : bons algorithmes de tri)
    - Les algo qui prennent un temps polynomial ne sont utilisables que pour des données de très petite taille

- Algorithme de recherche séquentielle dans une liste non triée :
    - Soit *liste* une liste non triée de longueur n, de type list[elem], et *e* un élément de type elem. On cherche s'il existe un indice $i \in [0, n - 1]$ tel que liste[i] == e
        - Parcourir la liste : pour tout $i \in [0, n - 1]$, faire :
            - Si liste[i] == e, retourner True
            - Sinon, si i == n - 1, retourner False
            - Sinon, i = i + 1

```python
def recherche_sequentielle_liste_non_triee(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste non triée"""
    appartient = False
    i = 0
    n = len(liste)
    while i < n and not(appartient) :
        if liste[i] == elem :
            appartient = True
        i = i + 1
    return appartient
```

- Complexité dans le meilleur des cas :
  - Opérations significatives : liste[i] == elem et n = len(liste)
  - Dépend de elem, de liste et de n
  - Complexité dans le meilleur des cas :
    - Si liste[0] == elem, 1 seul comparaison
- Complexité dans le pire des cas :
  - Opérations significatives : liste[i] == elem et n = len(liste)
  - Dépend de elem, de liste et de n
  - Complexité dans le pire des cas : Si liste [n – 1] == elem, ou elem $\in /$ liste, n comparaisons
- Complexité moyenne :
  - Soit q = p(elem $\in$ liste), et 1 – q = p(elem $\in /$ liste)
  - Nombre de comparaisons si elem $\in$ liste :
    - On suppose que la place de elem dans liste est équiprobable. Donc p(elem == liste[i]) = 1/n
    - Si elem == liste [i], il faut faire i + 1 comparaisons
    - Nombre moyen de comparaisons si elem $\in$ liste :

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

    - Nombre de comparaisons si elem $\in /$ liste : n comparaisons
    - Complexité moyenne :

$$\text{coût } moy(n) = q\frac{n+1}{2} + (1-q)n$$

    - Si q = 1/2, coût moy(n) = $\frac{3n+1}{4}$ Donc la complexité moyenne est de l'ordre de 3n/4

- Algo de recherche séquentielle dans une liste triée :

- Vérificateur de si la liste est triée :
  - Une *liste* de longueur n, de type list[elem] est triée si et seulement si :

```
def verif_liste_triee(liste):
    """List --> Bool.
    Vérifie si la liste est triée"""
    trie = True
    i = 0
    n = len(liste)
    while i<(n - 1) and trie :
        if liste[i] > liste[i+1]:
            trie = False
        i = i + 1
    return trie
```

    - $\forall i \in [0, n-2]$, liste[i] ≤ liste[i + 1]
    - Si n = 0 ou n = 1, la liste est triée

- Recherche séquentielle dans la liste triée :
  - Soit *liste* une liste triée de longueur n, de type list[elem], et e un élément de type elem. On cherche s'il existe un indice $\in [0, n-1]$ tel que liste [i] == e
  - Si e > liste[n-1], retourner False
  - Sinon, parcourir la liste : tant que liste[i] < e, faire i = i+1
  - A la sortie de la boucle,
    - Si liste[i] == e, retourner True
    - Sinon, retourner False

```
def recherche_sequentielle_liste_triee(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste triée"""
    if elem > liste[len(liste) - 1] :
        return False
    else :
        i = 0
        while liste[i] < elem :
            i = i + 1
        if liste[i] == elem :
            return True
        else :
            return False
```

- Cette méthode est maladroite : efficace si elem est présent en début de liste ou est rapidement plus petit que les éléments de la liste. Autrement, il faut parcourir beaucoup d'éléments !
- Complexité vue en O(n/2) = Complexité de l'ordre de n/2(voir TD)

- Recherche dichotomique dans une liste triée :
  - 2 cas possibles :
    - inf ≤ sup : on pose med = [(inf+sup)/2]
      - e == liste[med], retourner True
      - e < liste[med], susp = med – 1
      - e > liste[med], inf = med + 1
    - inf > sup : retourner False
  - Conditions initiales : inf = 0, sup = n – 1

```
def recherche_dichotomique(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste triée"""
    appartient = False
    inf, sup = 0, len(liste) - 1

    while inf <= sup and not(appartient) :
        med = (inf + sup)//2
        if liste[med] == elem :
            appartient = True
        elif liste[med] > elem:
            sup = med - 1
        else :
            inf = med + 1
    return appartient
```

- Complexité en O(log n) = Complexité de l'ordre de log n

Cours 8 :

- Tri par sélection :
  - Soit *liste* une liste non triée, que l'on veut trier, de longueur n de type list[elem]
  - Pour tout i ∈ [0, n – 2]
    - On parcourt la liste en cherchant le plus petit élément de *liste* pour j ∈ [i, n – 1]
    - On échange ce minimum avec liste[i]
  - L'algorithme de tri par sélection va utiliser la procédure *echange*(liste,i,j)
    - Prend en entrée une liste et 2 indices

- Echange les éléments de la liste correspondant à ces deux indices
- Le type list étant mutable, il n'est pas utile de retourner la liste donnée en argument d'appel, elle est directement modifié par la procédure

```
def echange(liste, i, j):
    """List x Int x Int --> None
    Echange les élements de liste en position i et j"""

    elem = liste[i]
    liste[i] = liste[j]
    liste[j] = elem
```

- Opérations significatives :
  - Comparaison de 2 éléments de la liste
  - Echange de deux éléments de la liste

```
def tri_selection(liste) :
    """List --> None -- Trie la liste donnée en paramètre.
    La liste est directement modifiée par la procédure."""
    n = len(liste)
    for i in range(n-1):
        indice_min = i
        for j in range(i+1, n):
            if liste[j] < liste[indice_min]:
                indice_min = j
        if indice_min != i:
            echange(liste, i, indice_min)
```

- Le nombre de comparaisons ne dépend pas des données de la liste à trier. Tous les cas sont équivalents en terme de complexité.
- Le nombre d'échanges dépend de la liste à trier
- Complexité pour les comparaisons :
  - i = 0 ; boucle de j = 1 à j = n – 1 → (n-1) comparaisons
  - i = 1 ; boucle de j = 2 à j = n – 1 → (n-2) comparaisons
  - …
  - i = n – 3 ; boucle de j = n – 2 à j = n – 1 → 2 comparaisons
  - i = n – 2 ; boucle de j = n – 1 à j = n – 1 → 1 comparaisons
    - $(n-1) + (n-2) + (n-3) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$ comparaisons
  - Complexité de l'ordre de $o(n^2)$
- Complexité pour les échanges :
  - Meilleur cas : la liste est déjà triée, on ne fait aucun échange

        - Pire cas : on fait un échange à chaque tour de boucle

- Tri par insertion :
    - Algorithme qu'utilise naturellement l'être humain pour trier des objets, comme par exemple des cartes à jouer
    - Soit *liste* une liste non triée, que l'on veut trier, de longueur n et de type list[elem]
    - Soit i $\in$ [1,n – 1], à l'étape i :
        - On suppose que les éléments d'indice 0 à i – 1 sont déjà triés
        - On insère l'élément d'indice i à sa place dans la liste *liste*[0: i – 1]
            - pos = i, sauvegarde de elem = liste[i]
            - Tant que liste[pos – 1] > elem, liste[pos] = liste[pos – 1] ; pos = pos – 1
            - Si pos == 0 ou liste[pos – 1] ≤ elem, liste[pos] = elem

```
def tri_insertion(liste):
    """List --> None
    Tri la liste donnée en paramètre"""
    for i in range(1, len(liste)):
        elem = liste[i]
        pos = i
        while pos > 0 and liste[pos - 1] > elem :
            liste[pos] = liste[pos-1]
            pos = pos - 1
        liste[pos] = elem
```

    - Opérations significatives :
        - Comparaison de deux éléments de la liste
        - Affectations d'un élément de la liste à elem, ou d'un élément de la liste à un autre
    - Le nombre de comparaisons et d'affectations dépend de la liste à trier
    - Complexité vu en TD

- Tri par comptage :
    - Principe : déterminer pour chaque élément de liste le nombre d'éléments qui lui sont inférieurs ou égaux
    - Pour trouver ind(i), 0 ≤ i ≤ n – 2, donnant la position de liste[i] dans la liste triée, on compare liste[i] à tous les liste[j], j $\in$ [i+1, n – 1]
        - Soit liste[j] ≤ liste[i] : on incrémente ind(i) de 1
        - Soit liste [j] > liste[i] : on incrémente ind(j) de 1

```
def tri_comptage(liste):
    """List --> List
    Tri la liste donnée en paramètre"""
    ind = []
    result = []
    n = len(liste)
    # initialisation des listes indices et résultat
    for i in range(n) :
        ind.append(0)
        result.append(0)
    for i in range(n - 1): #comptage
        for j in range(i+1, n) :
            if liste[j] > liste[i] :
                ind[j] = ind[j] + 1
            else :
                ind[i] = ind[i] + 1

    for i in range(n) : #liste triée résultat
        result[ind[i]] = liste[i]

    return result
```

- Opérations significatives :
  - Comparaison de deux éléments de la liste
  - Affectations d'un élément à la liste ind, et à la liste result
- Le nombre de comparaisons et d'affectations ne dépend pas de la liste à trier. Tous les cas sont équivalents en terme de complexité
- Comparaisons :

```
for i in range(n - 1): #comptage
    for j in range(i+1, n) :
        if liste[j] > liste[i] :
```

  - Pour i = 0, j va de 1 à (n – 1) → n – 1 comparaisons
  - Pour i = 1, j va de 2 à (n – 1) → n – 2 comparaisons
  - …
  - Pour i = n – 2, j va de (n – 1) à (n – 1) → 1 comparaison
    - (n – 1) + (n – 2) +...+ 2 + 1 = $\frac{n(n-1)}{2}$ comparaisons
    - Complexité de l'ordre de $O(n^2)$
- Affectations :

```
(1) for i in range(n) :
        ind.append(0)
        result.append(0)
(2) for i in range(n - 1): #comptage
        for j in range(i+1, n) :
            if liste[j] > liste[i] :
                ind[j] = ind[j] + 1
            else :
                ind[i] = ind[i] + 1
(3) for i in range(n) : #liste triée résultat
        result[ind[i]] = liste[i]
```

- (1) n append pour ind, n append pour result → 2n affectations
- (2) Autant d'affectations que de comparaisons → $\frac{n(n-1)}{2}$ affectations
- n affectations
  - Complexité de l'ordre de $o(n^2 + 3n)$

- Algorithme du drapeau à 3 couleurs :
  - Soit *liste* une liste non triée de longueur n contenant des données de 3 types : Rouge, Bleu et Jaune. On veut trier la liste de façon à ce que les premiers éléments soient bleus, les suivants jaunes, puis enfin les derniers rouges
  - Deux cas possibles
    - i = r + 1. C'est fini, la liste est triée
    - i ≤ r, 3 cas possibles
      - liste[i] == J, i = i + 1
      - liste[i] == B, echange(liste, j, i) ; i = i + 1 ; j = j + 1
      - liste[i] == R, echange(liste, r, i) ; r = r – 1

```
def drapeau(liste):
    """List --> None
    Tri la liste, contenant 3 couleurs R, J, B, donnée en paramètr
    i = 0
    j = 0
    r = len(liste) - 1
    while i <= r:
        if liste[i] == "J":
            i = i + 1
        elif liste[i] == "B" :
            echange(liste, j, i)
            i = i + 1
            j = j + 1
       else :
           echange(liste, r, i)
           r = r - 1
```

Cours 9 :

- Insertion d'un élément à sa place dans une liste triée :
  - *append* insère un élément à la fin d'une liste
  - Nous voulons modifier la liste en ajoutant un élément à sa place, ne pas en créer une nouvelle contenant cet élément et ne pas avoir besoin de trier de nouveau toute la liste…
  - Utiliser la méthode d'insertion déjà vu dans l'algorithme de tri par insertion :
    - append
    - Décaler les valeurs de la liste vers la droite, jusqu'à trouver la place de l'élément à insérer

```
def insertion(liste, elem):
    """List x Elem --> None
    Insère l'élément elem à sa position dans la liste triée"""
    liste.append(elem)
    n = len(liste)
    indice = n - 1
    while indice > 0 and liste[indice - 1] > elem :
        liste[indice] = liste[indice - 1]
        indice = indice - 1
    liste[indice] = elem
```

- Suppression d'un élément dans une liste
  - En algo :
    - Rechercher l'élément (voir progs cours 6)
    - Suppression de l'élément
    - La méthode de suppression dépend du langage de programmation
  - En Python :
    - *remove* supprime la 1ère occurrence de l'élément donné en paramètre
      - Pour supprimer toutes les occurrences : appliquer *remove* tant que l'élément appartient à la liste
    - *del* supprime l'élément positionné à l'indice donné en paramètre
    - Une autre possibilité est d'utiliser les compréhensions de listes :

```
>>> liste = ['a', 'b', 'c', 'a', 'b', 'd']
>>> liste = [elem for elem in liste if elem != 'b']
>>> liste
['a', 'c', 'a', 'd']
```

- Compréhension de listes
  - Différence entre définition explicite et définition implicite/par compréhension :
    - E = {1, 2, 3, 4, 5}, définition explicite
    - $E = \{n \in IN^* \}$, définition implicite/par compréhension
  - On aimerait pouvoir exprimer littéralement en python, ex : Construire la liste des i pour $i \in [1,5]$ => Construction par compréhension

- Syntaxe d'une construction par compréhension :

```
[<expr> for <var> in <seq>]
```

  - <var> : une variable de compréhension
  - <expr> : une expression pouvant contenir <var>
  - <seq> : une séquence (range, str ou list)
  - Construit la liste composée des éléments :
    - Le premier élément est la valeur de <expr> dans laquelle la variable <var> a pour valeur
    - Le deuxième élément est la valeur de <expr> dans laquelle la variable <var> a pour valeur le deuxième élément de <seq>
    - …
    - Le dernier élément est la valeur de <expr> dans laquelle la variable <var> a pour valeur le dernier élément de <seq>
  - Quelques exemples simples :

```
def naturels_comprehension(n) :
    """Int --> List.
    Retourne la liste des n premiers entiers non nuls"""

    return [i for i in range(1, n+1)]
```

```
def liste_carres_comprehension(liste) :
    """List --> List.
    Retourne la liste des carrés des éléments
    de la liste donnée en paramètre"""

    return [elem * elem for elem in liste]
```

```
def liste_longueurs_comprehension(liste) :
    """List --> List.
    Retourne la liste des longueurs des chaines
    de la liste donnée en paramètre"""

    return [len(s) for s in liste]
```

- Schéma de filtrage :
  - Le filtrage d'une liste retourne une sous-liste de la liste de départ, selon un prédicat donnée

```
def liste_positifs(liste) :
    """list --> list
    Retourne la sous-liste des entiers positifs
    de la liste donnée en paramètre"""

    resultat = []
    for elem in liste :
        if elem > 0:
            resultat.append(elem)
    return resultat
```

- Syntaxe d'une construction par compréhension conditionnée :

```
[<expr> for <var> in <seq> if <condition>]
```

  - <var>, <expr>, <seq> comme dans une compréhension classique
  - <condition> : une expression booléenne portant sur <var>
  - Construit la liste de la même façon, en ne retenant que les éléments pour lesquels la <condition> est True

```
def liste_pairs(liste) :
    """List --> List.
    Retourne la sous-liste des entiers pairs
    de la liste donnée en paramètre"""

    return [elem for elem in liste if elem % 2 == 0]
```

```
def longueur_min(liste) :
    """List --> List.
    Retourne la sous-liste des chaines de la liste
    donnée en paramètre contenant au moins 3 caractères"""

    return [s for s in liste if len(s) >= 3]
```

- Utilisation du if…else dans une construction par compréhension ?
  - On met le if…else avant le for, contrairement à quand il n'y a pas de else

```
>>> liste = [2, 3, 4, 5, 6]
>>> [elem**2 if elem % 2 == 0 else elem**3 for elem in liste]
[4, 27, 16, 125, 36]
```

- Syntaxe d'une construction par compréhension conditionnée
  - <var>, <seq>, <condition>
  - <expr1> : une expression pouvant contenir <var>
  - <expr2> : une expression pouvant contenir <var>
  - Si <condition> est True, <expr1> sera appliqué ; autrement <expr2> sera appliqué

- Compréhensions multiples :
  - Problème : Donner une définition de la fonction qui, étant donné un entier n, renvoie la liste des couples (i, j) dans l'intervalle [1, n] avec i ≤ j
- Syntaxe d'une construction par compréhension multiple :

```
[<expr> for <var1> in <seq1> for <var2> in <seq2> ... ]
```

- Syntaxe d'une construction par compréhension complète :

```
[<expr> for <var1> in <seq1> if <cond1>
        for <var2> in <seq2> if <cond2>
        ... ]
```

```
>>> liste_couple(3)
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

```
def liste_couples(n) :
    """Int --> List, avec n >= 0
    Retourne la liste des couples (i,j) sur l'intervalle
    [1, n] avec i <= j"""

    liste = []
    for i in range(1, n + 1) :
        for j in range(i, n+1) :
            liste.append((i, j))
    return liste
```

Cours 10 :

- IndexError, NameError, ZeroDivisionError sont des exceptions
- Lorsque le programme rencontre une erreur, il s'arrête brutalement et affiche la trace d'exécution
- L'instruction assert sert à lever une exception
  - Permet d'avoir des messages d'erreur plus lisibles

```
def moyenne(liste) :
    """List --> Float. Calcule la moyenne de la liste"""
    assert len(liste) > 0, "Moyenne d'une liste vide"
    return somme(liste)/len(liste)
```

```
>>> moyenne([])
AssertionError: Moyenne d'une liste vide
```

- Gestion des exceptions :
  - try :
    <sequence_instructions1>
    except :
    <sequence_instructions2>
  - Si au cours de l'exécution de la sequence_instructions1 une exception se produit, l'exécution du bloc est abandonnée et la sequence_instructions2 est exécutée. Si l'exécution de sequence_instructions1 s'est déroulée normalement, on ne rentre pas ds except

- Il est possible de ne vouloir gérer que certaines exceptions :

- Il y a différents types d'exceptions : `IndexError`, `NameError`, `ZeroDivisionError`...
- Il est possible de ne vouloir gérer que certaines exceptions, ou de faire des traitements différents en fonction de du type d'erreur rencontrée

```
try :
    <sequence_instructions1> #séquence normale d'exécution
except <type_exception1> :
    <sequence_instructions2> #traitement de l'exception 1
except <type_exception2> :
    <sequence_instructions3> #traitement de l'exception 2
...
else :
    #bloc d'instruction exécuté en l'absence d'erreurs
    <sequence_instruction4>
```

- Fichier : collection d'informations stockées sur une mémoire de masse (disque dur, clef usb, CD, …). On y accède grâce à son nom précédée, si besoin, du chemin d'accès vers le fichier
  - Pour simplifier, dans ce cours :
    - On suppose que tous les fichiers manipulés se trouvent de le répertoire courant
    - On ne manipule que des fichiers textes
  - 2 opérations distinctes sur les fichiers :
    - On peut vouloir écrire des données dans un fichier : accès en écriture
    - On peut vouloir lire des données à partir d'un fichier : accès en lecture
- open() : est une fonction intégrée qui permet de créer un objet-fichier
  - fonction intégrée car on l'exécute ainsi :
    mon_fichier = open('Monfichier.txt', 'a')
  - open() attend 2 arguments sous forme de str :
    - Le nom du fichier à ouvrir
    - Le mode d'ouverture
  - L'option 'a' permet d'ouvrir le fichier en mode ajout (append)
    - S'il existe un fichier du nom indiqué, les données enregistrées sont ajoutées à la fin du fichier
    - S'il existe un fichier du nom indiqué, un nouveau fichier est crée
  - L'option 'w' permet d'ouvrir le fichier en mode écriture (write)

        - S'il existe un fichier du nom indiqué, ce fichier est écrasé et l'écriture des données commence à partir du début d'un nouveau fichier, vide
        - S'il n'existe pas de fichier de ce nom, un nouveau fichier est créé
    - L'option 'r' permet d'ouvrir le fichier en mode lecture (read)
        - S'il n'existe pas de fichier de ce nom, une exception est levée : FileNotFoundError

```
mon_fichier = open('Monfichier.txt', 'a')
mon_fichier.write('Bonjour !\n')
mon_fichier.write('Saperlipopette ! ')
mon_fichier.write('Dit le poète ! ')
mon_fichier.close()
```

    - La méthode write() écrit les données voulues dans le fichier
        - Les données sont enregistrées les à la suite des autres
        - Si le caractère de retour à la ligne \n n'est pas indiqué, le prochain write() se fera sur la même ligne
    - La méthode close() ferme le fichier
        - Les écritures sont mises en tampon, elles ne prennent pas forcément effet immédiatement. Elles peuvent ne pas êtres enregistrées tant que le fichier n'est pas fermé !
        - Donc il ne faut pas oublier le close()
    - La méthode write() ne permet d'écrire que des chaînes de caractères. Pour écrire un nombre, il faut donc le transformer en str d'abord :

```
>>>mon_fichier = open('Monfichier.txt', 'a')
>>>mon_fichier.write(1975)
TypeError: write() argument must be str, not int
>>>mon_fichier.write(str(1975))
```

    - La méthode read() lit la totalité du fichier dans une seule chaîne de caractères
        - Cette méthode peut également être utilisée avec un argument qui indique le nombre de caractères qui doivent être lus, à partir de la position déjà atteinte dans le fichier
    - La méthode readline() permet de lire le fichier ligne à ligne

- Exemple : générer un fichier contenant une table de multiplication

```
def table_fichier(n) :
  """Int --> None
  Créé un fichier contenant la table de multiplication de n"""

  #Créer le fichier
  f_table = open("table" + str(n) + ".txt", "w")

  #Ecriture dans le fichier
  for i in range(1, 11):
    f_table.write(str(i) + '*' + str(n) + '=' + str(i*n) + '\n')

  #Fermeture du fichier
  f_table.close()
```

```
>>> table_fichier(n)
```

```
more table7.txt
1*7=7
2*7=14
3*7=21
4*7=28
5*7=35
6*7=42
7*7=49
8*7=56
9*7=63
10*7=70
```

-