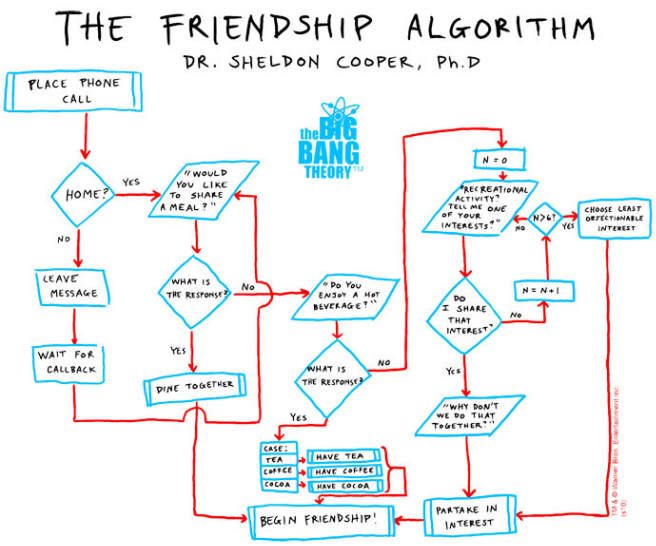
On aborde dans ce chapitre l'étude théorique des algorithmes.



1. Les algorithmes et les problèmes spécifiques qu'ils soulèvent

Voici la définition d'un algorithme que l'on peut lire dans le Larousse :

> *Ensemble de règles opératoires dont l'application permet de résoudre un problème énoncé au moyen d'un nombre fini d'opérations.*

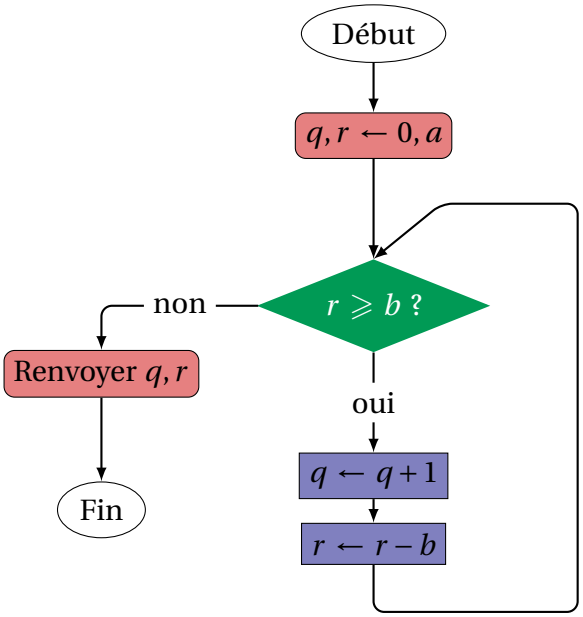
Il existe quatre principales manières de décrire un algorithme. Nous les illustrerons au moyen de la division euclidienne de a par b par soustraction successives (avec $b \in \mathbb{N}^*$ et $a \in \mathbb{N}$) :

✕ Description sommaire et informelle dans le langage courant.

⇛ Tant que le résultat est supérieur à b, on retranche b à une variable valant initialement a.

⇛ En fin d'algorithme, le nombre d'itérations est le quotient, la valeur de la variable est le reste.

✕ Donnée d'un organigramme.



✕ Donnée d'un code.

Ici nous avons choisi un code dans le langage Python.

```
def euclide(a,b):
    q,r=0,a
    while r>=b:
        q,r=q+1,r-b
    return [q,r]
```

Introduit sous cette forme, il est important d'utiliser des noms de variables explicites (par exemple q comme quotient, r comme reste) voire d'ajouter des commentaires au sein du code afin de faciliter la compréhension de l'algorithme (au moyen du caractère # en Python).

✕ Donnée d'un pseudo-code.

Un pseudo-code est une description intermédiaire entre le langage courant et les langages de programmation. On peut le lire comme une recette de cuisine, qu'un programmeur peut ensuite traduire dans n'importe quel langage. Il faut comprendre $x \leftarrow x+u$ comme l'affectation $x = x+u$ sous Python. Les mots-clés *tant que*, *pour* et *si ... alors ... sinon ...* remplacent les mots clés *while*, etc.

⇛ Données : Deux entiers naturel a et $b > 0$.

⇛ Résultat : Couple quotient-reste dans la division de a par b.

⇛ Initialisation, : $q \leftarrow 0$, $r \leftarrow a$.

⇛ Tant que $r \geqslant b$:

- $q \leftarrow q+1$
- $r \leftarrow r-b$

⇛ Renvoyer la liste $[q, r]$.

Voici trois questions essentielles auxquelles nous apporterons des éléments de réponse :

✕ Comment démontrer qu'un algorithme s'arrête ?

Un algorithme utilisant une boucle `while` peut ne jamais se terminer à cause d'une *boucle infinie*. Il faut savoir dans ce cas *démonter* qu'elle s'arrête.

✕ Comment démontrer qu'un algorithme est correct ?

Quand un algorithme s'arrête, le résultat renvoyé est-il le résultat attendu ? Autrement dit, l'algorithme est-il correct ? La réponse à cette question n'est pas toujours aisée à donner.

✕ Comment comparer des algorithmes ?

On peut souvent imaginer plusieurs algorithmes pour résoudre un problème donné. Un critère important de choix est leur rapidité d'exécution.

2. Conseils de programmation

Après un semestre de TP rondement menés, nous pouvons rassembler quelques recommandations pour la conception et le codage d'une fonction en Python.

Commençons par un peu de vocabulaire :

⇛ UNE EXPRESSION est utilisée pour un calcul. Par exemple, `x**2-2*x+1` est une expression (*x* désignant une variable). Toute expression a une valeur. Ainsi, l'expression `x==1` vaut `True` ou `False` (c'est une expression booléenne).

⇛ UNE AFFECTATION est utilisée pour associer un nom de variable à une valeur (ou modifier un terme d'une liste par exemple).

⇛ UNE INSTRUCTION est une portion de code minimale qui produit un effet. Les affectations et les expressions sont des instructions, c'est aussi le cas de `assert` (cf. plus loin), `return`, `break` ou encore `import`. Tout langage dispose aussi d'instructions composées (tests, boucles par exemple).

On dit qu'une fonction présente un effet de bord (*side effect* in English) si un appel à cette fonction peut modifier l'une de ses variables d'entrée.

Règle d'or pour les concours

Sauf mention du contraire dans l'énoncé, on évitera tout effet de bord dans une fonction.

Il faut en particulier différencier les cas où le sujet demande une fonction devant *renvoyer une expression*, et ceux où la fonction doit se contenter de *modifier une (ou plusieurs) de ses variables*. Certains auteurs appellent *procédures* les secondes.

Sous Python, la gestion des variables lors d'un appel de fonction ne permet d'effet de bord que sur les variables de type composé (les listes, les tableaux `ndarray` mais pas les types numériques `int` ou `float`). Par exemple, nous avons implémenté des tris en place, i.e. par permutation, en utilisant un effet de bord. C'est en général le cas pour ce type d'algorithme : une fois le tri accompli, la liste initiale est sans intérêt pour les applications futures.

2.1. Conception et implémentation d'un algorithme

✕ Concevoir un algorithme.

Avant de coder quoi que ce soit, il faut réfléchir un peu au brouillon afin de déterminer une stratégie de résolution et une méthode de programmation (itérative, récursive). En particulier, il faut bien choisir ses variables (en nombre minimal) et connaître clairement leur sens ainsi que leur rôle dans l'algorithme. Nous verrons un peu plus loin dans ce cours que la notion *d'invariant de boucle* peut s'avérer une aide précieuse.

✕ Implémenter un algorithme en Python (sur machine ou sur feuille).

Comme en Mathématiques, il faut avant tout que le code d'une fonction soit concis et clair. À ce titre, on évitera :

- ✔ Les variables inutiles (par exemple l'utilisation de deux variables dont la somme est constante) ;
- ✔ Les noms de variables obscurs (plutôt que x et y, il convient d'utiliser q et r pour un quotient et un reste).

On prendra soin de :

- ✔ Ajouter des commentaires aux lignes du programme (ce qui est écrit sur une même ligne après le caractère # est ignoré à l'exécution) ;
- ✔ Ajouter dans une copie une description succincte de l'algorithme et la signification des variables (ce qu'elles représentent et leur rôle dans l'algorithme).

2.2. Notice d'une fonction

Sous Python, le code d'une fonction peut contenir une chaîne de caractères (*docstring* en anglais) qui permettant de la documenter, de détailler les types de ses paramètres d'entrée et de ses sorties.

```
def maxrang(t,k):
    """
    Renvoie le maximum de la sous-liste [t[0],...,t[k]] :
    ---
    Entrees :
              t : liste d'entiers
              k : entier
    Sortie :  un entier
    Exemple : maxrang([2,1,4,3,6,4],2) renvoie 4
    """
    ind=0
    for i in range(1,k+1):
        if t[i]>t[ind]:
            ind=i
    return t[ind]
```

Dans cette notice, il convient de préciser :

- le but de la fonction, avec ses éventuelles limitations ;
- les données en entrée (type, valeurs) et les données en sortie (i.e. renvoyées par la fonction) ;
- des exemples significatifs d'utilisation.

Cette chaîne est utilisée par la commande `help(fonction)` :

```
help(maxrang)
Help on function maxrang in module \_main\_:
maxrang(t,k)
            Renvoie le maximum de la sous-liste [t[0],...,t[k]] :
            ---
            Entrees :
                  t : liste d'entiers
                  k : entier
            Sortie :   un entier
            Exemple : maxrang([2,1,4,3,6,4],2) renvoie 4
```

2.3. Assertions

La notice d'une fonction ne fournit que des indications sur les arguments de la fonction. Si elle est appelée avec des arguments d'un mauvais type, elle peut aussi bien provoquer une erreur (ce qui est le mieux qui puisse arriver) ou, pire, continuer en renvoyant un résultat incohérent (parce que les opérations internes à la fonction ne provoquent pas d'erreurs).

On peut y remédier facilement au moyen de la fonction `assert`.

```
def maxrang(t,k):
    """
    Renvoie le maximum de la sous-liste [t[0],...,t[k]] :
    ---
    Entrees :
              t : liste d'entiers
              k : entier
    Sortie :   un entier
    Exemple :  maxrang([2,1,4,3,6,4],2) renvoie 4
    """
    assert(type(t)==list), 't doit etre une liste'
    assert(type(k)==int), 'k doit etre un entier'
    assert(all([type(x)==int for x in t])), 't doit ne contenir que des entiers'
    assert(0<=k<len(t)), 'indice k non valide'
    ind=0
    for i in range(1,k+1):
        if t[i]>t[ind]:
            ind=i
    return t[ind]


assert(maxrang([],0)==None)
assert(maxrang([4,2,0],2)==4)
assert(maxrang([4,4,0],2)==4)
```

```
assert(maxrang([1,2,4],2)==4)
assert(maxrang("23456",2)==None)
```

On voit que l'on peut ainsi contrôler les entrées choisies par l'utilisateur de la fonction et provoquer un message d'erreur ad hoc.

```
maxrang(t,78)
AssertionError: indice k non valide
```

2.4. Jeu de tests associé à un programme

Il n'est pas toujours simple de justifier qu'une fonction renvoie *dans tous les cas* la valeur attendue. À défaut d'une démonstration (cf. plus loin dans ce cours), on peut se contenter de tester le code sur certains exemples bien choisis, notamment en s'intéressant aux cas extrêmes.

Considérons l'exemple d'une fonction `max` renvoyant le maximum d'une liste d'entiers. On peut envisager :

- de la tester sur une liste « standard » ;
- de la tester sur quelques cas extrêmes : est-elle correcte si le maximum est à l'une des extrémités (début ou fin), s'il y a plusieurs indices pour le maximum ? si la liste n'a qu'un élément ?
- de la tester dans des cas « exclus » : celui de la liste vide par exemple.

Par exemple, la fonction suivante passe le jeu de tests décrit ci-dessous :

```
def max(t):
    if len(t)==0:
        return None
    ind=0
    for i in range(1,len(t)):
        if t[i]>t[ind]:
            ind=i
    return t[ind]
```

```
assert(max([])==None)
assert(max([2])==2)
assert(max([2,3])==3)
assert(max([2,6,7,8,3,2,11,4])==11)
assert(max([4,2,3,0])==4)
assert(max([1,9,5,2,4])==9)
assert(max([1,8,5,8,4])==8)
```

Plus généralement, il y a deux idées principales lorsqu'on veut réaliser un jeu de tests sur un programme :

- **Partionnement** : l'idée est de ne pas tester toutes les entrées possibles mais d'en faire un partitionnement, c'est-à-dire écrire l'ensemble des données d'entrée E en une réunion finie de sous-ensembles disjoints $E = E_1 \cup E_2 \cup \cdots \cup E_k$ et de vérifier que pour chaque sous-ensemble le programme est correcte sur au moins une valeur.
- **Tests aux limites** : vérifier que le programme est correct sur certaines valeurs limites.

Dans l'exemple précédent, on a partitionné l'ensemble des listes en trois sous-ensembles : les listes à zéro, un et au moins deux éléments. Les cas limites sont les cas où le maximum est aux extrémités et celui où il apparaît deux fois.

3. Terminaison et correction d'un algorithme

Nous abordons ici la question de la *validité* d'un algorithme. On peut décomposer la démonstration de la validité d'un algorithme en deux étapes :

- *prouver qu'il s'arrête* : les boucles `for` s'arrêtant par définition, cela revient à justifier que toutes les boucles `while` s'arrêtent.
- *prouver qu'il est correct* : vérifier quà la fin de l'algorithme, la valeur renvoyée est bien celle qui était attendue.

3.1. Variant de boucle et terminaison

Comme mentionné ci-dessus, nous ne considérons dans ce paragraphe que des boucle conditionnelles, i.e. des boucles `while`.

Définition 2.0. Variant de boucle

Un variant de boucle est une expression V, dépendant des variables du programme et à valeurs dans $\mathbb{N}$, qui décroit strictement d'une itération à l'autre.

L'intérêt d'un variant est de garantir la *terminaison*[1] d'une boucle, i.e. le fait qu'elle ne soit pas infinie. Cette stratégie de démonstration est appelée *méthode de Floyd*. Elle repose sur le fait qu'il n'existe aucune suite d'entiers naturels strictement décroissante.

Proposition 2.1. Terminaison de boucle avec variant

Si une boucle admet un variant, alors elle termine nécessairement.

```
def euclide(a,b):
    q,r=0,a
    while r>=b:
        q,r=q+1,r-b
    return [q,r]
```

Considérons par exemple, la fonction ci-contre qui renvoie, sous la forme d'une liste, le couple quotient-reste dans la division euclidienne de deux entiers naturels a et b (avec b non nul).

Il est clair que la variable `r` est un variant de la boucle de cette fonction.

1. Ce terme, directement issu de l'anglais, est couramment utilisé en français.

Voici l'exemple célèbre de la suite de Syracuse. On conjecture que, pour tout $u_0 \in \mathbb{N}$, il existe $n_0 \in \mathbb{N}$ tel que $u_{n_0} = 1$.

Cette proposition n'est à ce jour ni démontrée, ni infirmée.

La suite de Syracuse est définie par

$$u_0 \in \mathbb{N}, \quad \forall n \in \mathbb{N}, \quad u_{n+1} = \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

Cela signifie que personne n'a encore réussi à prouver ni à infirmer la terminaison de la fonction ci-contre.

```
def syr(a):
    u0,n=a,0
    while a>1:
        if a%2==0:
            a=a//2
        else:
            a=3*a+1
        n=n+1
    return n
```

Le lecteur poursuivra avec le test (⚡ 2.1).

3.2. Invariants de boucle

Les invariants de boucle sont un outil efficace pour prouver la correction d'un algorithme.

Définition 2.2. Invariant de boucle

On appelle invariant de boucle une propriété vérifiée par les variables à la fin de chaque chaque itération de la boucle.

Un invariant se justifie au moyen d'un raisonnement par récurrence : comme en Mathématiques, il faudra parfois augmenter (cf. les récurrences fortes ou doubles) la propriété à démontrer pour rendre le raisonnement par récurrence possible.

Revenons à la fonction `euclide` décrite au paragraphe précédent. À chaque itération de la boucle, la variable `q` augment d'une unité alors `r` diminue de b, ainsi l'expression `qb+r` est constante au fil des itération. Comme elle vaut initialement a, on en déduit que $a = qb + r$ est un invariant de boucle. En ajoutant la propriété $(q, r) \in \mathbb{N}^2$, clairement invariante, on obtient un invariant de boucle permettant de démontrer la correction de cet algorithme :

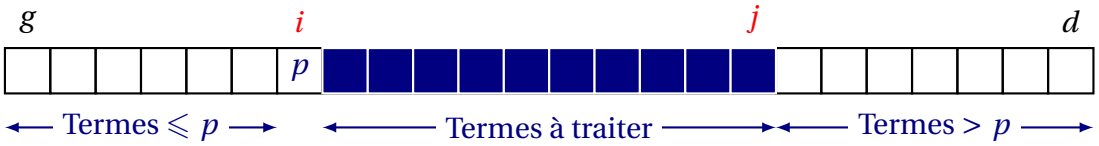
- ✗ la boucle termine (cf. le paragraphe précédent) ;
- ✗ lorsque la boucle s'arrête, $r < b$ et comme $r \geqslant 0$ et $a = bq + r$, le couple (q, r) est bien le couple quotient-reste dans la division euclidienne de a par b.

Finalement, lors de l'a conception d'un algorithme, les variables que nous utilisons ont un sens précis le plus souvent lié à un invariant.

- ✗ Revenons par exemple sur la méthode de segmentation selon un pivot utilisée pour l'implémentation du tri rapide (cf. le TP dédié aux tris). Pour effectuer cette *segmentation* en place, l'idée est d'effectuer un seul parcours de la liste entre les indices `g` et `d` en préservant la propriété suivante :

$$t[i] = p, \quad \forall \ell \in [\![g, i[\![, \ t[\ell] \leqslant t[i] \quad \text{et} \quad \forall \ell \in]\!]j, d]\!], \ t[\ell] > t[i]$$

  On utilise ainsi deux indices i et j vérifiant la propriété précédente. On initialise par $(i, j) = (g, d)$. En fin d'algorithme $j = i$, et la segmentation est accomplie.



La propriété définissant les variables i et j est un invariant de boucle.

- Voir le test (⚡ 2.2) pour un exemple élémentaire.
- Le lecteur s'entraînera au moyen du test (⚡ 2.3) sur le codage en base 2.

Utilisation d'un invariant de boucle pour concevoir un algorithme

On peut s'aider d'un invariant pour concevoir un algorithme. En définitive, on ne comprend *vraiment* un algorithme que si l'on est capable de trouver un invariant qui justifie sa correction.

On pourra mettre en pratique la recherche d'un invariant en vue d'une démonstration de correction dans le test (⚡ 2.4).

3.3. Cas d'une fonction récursive

Dans le cas d'une fonction récursive, on pourra opter pour une démonstration par récurrence de la terminaison et de la correction. Considérons par exemple, la version récursive de l'algorithme de la division euclidienne étudié dans les paragraphes précédents.

On démontre le résultat par récurrence sur a. Il est clair, par l'initialisation de la fonction, que l'hypothèse est correcte à tous les rangs de 0 à $b-1$.

Soit $a \geqslant b-1$. Supposons la propriété vraie jusqu'au rang a. Comme $a+1 \geqslant b$, la fonction fait l'appel récursive sur la couple $(a+1-b, b)$ avec $a+1-b \leqslant a$. Ainsi, par l'hypothèse au rang $a+1-b$, cet appelle termine et renvoie des entiers naturels q et r tels que $a+1-b = bq+r$ avec $r < b$. On en déduit que $a+1 = b(q+1)+r$ et donc que la fonction termine en renvoyant le bon couple.

```
def div(a,b):
    if a<b:
        return [0,a]
    else:
        [q,r]=div(a-b,b)
        return [q+1,r]
```

Correction d'une fonction récursive

Ce type de démonstration peut nous aider à créer une fonction récursive. Il faut se poser la question suivante : en supposant que l'appel récursif se solde par un résultat correct, comment conclure l'algorithme ?

4. Complexité d'un algorithme

Afin de comparer plusieurs algorithmes résolvant un même problème, on introduit des *mesures de performance* de ces algorithmes appelées *complexités* ou encore *coût*.

Deux critères se dégagent clairement : la *rapidité* d'un algorithme et *l'espace mémoire* nécessité par l'exécution d'un algorithme. De nos jours, la performance des composants électroniques rend moins important le second aspect par rapport au premier.

4.1. Complexité temporelle

L'idée est d'estimer le temps que prend l'exécution d'un algorithme en faisant l'hypothèse que les tâches élémentaires (comparaisons, additions, multiplications et affectations numériques) sont faites par le processeur avec un temps caractéristique constant τ. Actuellement, pour les processeurs les plus commercialisés, on a $\tau \simeq 10^{-9}\,s$, une nano-seconde.

Définition 2.3. Complexité ou coût temporel

On appelle complexité temporelle le nombre d'opérations élémentaires nécessitées par l'algorithme. Par opérations élémentaires, on entend :

les comparaisons, les affectations, les additions et les multiplications

Sous ce modèle, la complexité c est proportionnelle au temps d'exécution t : $t = c \times \tau$ et c est donc une mesure de la vitesse d'exécution de l'algorithme. Dans la pratique, ce modèle a des limites. Par exemple, le temps de calcul du résultat d'une addition d'entiers dépend clairement du nombre de décimales de ces entiers et son approximation par τ devient très grossière lorsque ce nombre explose.

La complexité temporelle dépend en général des données. Dans les cas simples, on peut exprimer cette complexité en fonction de la « *taille* » des données. La complexité est alors une suite $(C_n)_{n\in\mathbb{N}}$.

```
def fibo(n):
    if n<=1:
        return n
    u0,u1=0,1
    for i in range(n-1):
        u0,u1=u1,u0+u1
    return u1
```

On a $C_0 = C_1 = 1$ (on compte un test scalaire).

Pour $n \geqslant 2$, on trouve

$$C_n = 3 + 3 \times (n-1) = 3n$$

car on dénombre un test et deux affectations initiales puis $n-1$ itérations comptant chacune une opération et deux affectations.

Ce n'est pas toujours possible. Par exemple, la complexité de certains algorithmes sur les listes ne dépend pas seulement de la longueur n de la liste d'entrée mais aussi des termes de celle-ci. Considérons par exemple la recherche séquentielle de x dans une liste t de longueur n.

- ⇛ Dans le cas où x est le premier terme, la complexité vaut 1 (un seul test).
- ⇛ Dans le cas où x est en fin de liste, la complexité vaut n (n itérations comptant chacune un test).

```
def search(t,x):
    for a in t:
        if a==x:
            return True
    return False
```

On trouve non seulement des valeurs différentes mais des ordres de grandeurs différents.

Définition 2.4. Ordres de grandeurs usuels

Une complexité en :

- ⇛ $\Theta(1)$ est dite constante.
- ⇛ $\Theta(\ln n)$ est dite logarithmique.
- ⇛ $\Theta(n)$ est dite linéaire.
- ⇛ $\Theta(n \ln n)$ est dite quasi-linéaire.
- ⇛ $\Theta\left(n^2\right)$ est dite quadratique.
- ⇛ $\Theta\left(n^\beta\right)$ (avec $\beta > 1$) est dite polynomiale.
- ⇛ $\Theta\left(a^n\right)$ (avec $a > 1$) est dite exponentielle.

La notation $u_n = \Theta(v_n)$ est définie par $u_n = O(v_n)$ et $v_n = O(u_n)$, ce qui équivaut pour des suites positives à l'existence de $\lambda > 0$ et $\mu > 0$ telles que, à partir d'un certain rang :

$$\lambda\, v_n \leqslant u_n \leqslant \mu\, v_n$$

Afin de mieux aprécier ces ordres de grandeurs, considérons le cas d'un processeur effectuant 10^9 opérations par secondes. Les temps d'exécution sont donnés pour $n = 10^6$.

$\Theta(1)$	Temps constant	1 ns	Le temps d'exécution ne dépend pas de la donnée.
$\Theta(\ln n)$	Logarithmique	10 ns	Exécution quasi instantanée.
$\Theta(n)$	Linéaire	1 ms	Pour des données de la taille des mémoires vives, le temps d'exécution avoisine la minute. Le problème de gestion de la mémoire se posera donc *avant* celui du temps d'exécution.
$\Theta(n \ln n)$	Quasi-linéaire	14 ms	Voir ci-dessus, c'est encore raisonnable.
$\Theta\left(n^2\right)$	Quadratique	15 min	Acceptable pour $n \leqslant 10^6$.
$\Theta\left(n^k\right)$	Polynomiale	30 ans ($k = 3$)	Complexité très courante.
$\Theta\left(2^n\right)$	Exponentielle	$10^{2\,700\,000}$ ans.	On dépasse l'âge de l'univers ! C'est impraticable sauf pour $n \leqslant 50$.

C'est l'ordre de grandeur qui compte

Un calcul de complexité porte sur les ordres de grandeurs. Il convient d'estimer le coût de chacune des étapes de l'algorithme afin de déterminer sa complexité. Nous verrons plus loin quelques exemples éclairants de rédaction.

✕ Ainsi, la complexité de la fonction `fibo` de ci-dessus est $\Theta(n)$.

✕ Le lecteur poursuivra avec le test (↯ **2.5**).

Définition 2.5. Complexité dans le pire ou le meilleur des cas

Lorsque l'ordre de grandeur de la complexité d'un algorithme dépend de ses paramètres d'entrée, on peut en donner un « encadrement » en précisant :

- la complexité dans le pire des cas ;
- la complexité dans le meilleur des cas.

Convention pour les concours

En l'absence de précision, on entendra par « complexité », la complexité dans le pire des cas et on l'exprimera sous la forme d'un O (« grand O »).

✗ Dans le cas de la recherche séquentielle dans une liste, la complexité dans le meilleur des cas est $\Theta(1)$ et $\Theta(n)$ dans le pire des cas : dans tout les cas, la domination $O(n)$ est donc correcte.

4.2. Complexité des opérations sur les listes et les dictionnaires

Le tableau suivant regroupe les principales complexités des méthodes sur les listes. Elles sont à connaître sauf celles de l'insertion et la délétion qui sont données à titre informatif.

Action	Code	Coût
Ajout d'un élément `x` en tête de liste	`t.append(x)`	constant
Insertion d'un élément `x` en position `i`	`t.insert(i,x)`	linéaire
Renvoi et suppression du dernier terme	`t.pop()`	constant
Suppression du terme d'indice `i`	`del t[i]`	linéaire
Recherche de `x` dans `t`	`x in t`	linéaire
Longueur de la liste	`len(t)`	constant
Accès à la cellule n° `i`	`t[i]`	constant
Copie d'une liste à valeurs scalaires	`t1=t[:], t1=copy.copy(t)`	linéaire

Nous n'entrerons pas en détail sur l'implémentation en mémoire des listes de Python qui permettraient de justifier ces ordres de grandeurs. Il faut savoir que les complexités en temps constant du tableau ci-dessus sont en fait vraies en moyenne mais on les utilise tout de même comme valeurs de référence dans les calculs.

La structure de dictionnaire propose quelques optimisations en termes de complexité (ceci vient de son implémentation via des tables de hachage qui sera étudiée l'année prochaine).

Action	Code	Coût
Ajout d'une association	`d[cle]=valeur`	constant
Suppression de l'association de clé `cle`	`del d[cle]`	constant
Recherche de la clé `cle` dans `d`	`cle in d`	constant
Longueur du dictionnaire	`len(d)`	constant
Accès à la valeur de clé `cle`	`d[cle]`	constant
Copie d'un dictionnaire à valeurs scalaires	`d1=d[:], d1=copy.copy(d)`	linéaire

Listes Vs. dictionnaires

L'utilisation d'un dictionnaire est préférable à celle d'une liste dans des algorithmes nécessitant une recherche ou des délétions.

4.3. Complexités des algorithmes classiques

Nous allons revenir sur quelques algorithmes déjà rencontrés lors des séances de TP du premier semestre.

✕ L'algorithme de recherche linéaire.

Nous l'avons déjà étudié ci-dessus mais y revenons afin d'insister sur la rédaction. Notez-bien la différence avec la version précédente.

```
def search(t,x):
    for a in t:
        if a==x:
            return True
    return False
```

- ✔ Chaque itération est de coût constant.
- ✔ Le nombre d'itérations vaut au minimum 1 et au maximum n, longueur de la liste.
- ✔ La complexité totale est donc en $O(n)$. On peut préciser que la complexité dans le meilleur des cas vaut $O(1)$ (lorsque x est le premier terme de la liste).

✕ L'algorithme soustractif de division euclidienne.

```
def euclide(a,b):
    q,r=0,a
    while r>=b:
        q,r=q+1,r-b
    return [q,r]
```

- ✔ Chaque itération est de coût constant.
- ✔ Le nombre d'itérations est le quotient dans la division de a par b, i.e. $\left\lfloor \frac{a}{b} \right\rfloor$.
- ✔ La complexité totale est donc en $O\left(\frac{a}{b}\right)$.

✕ La recherche dichotomique dans une liste triée.

```
def bisearch(t,x):
    g,d=0,len(t)-1
    if x<t[g] or t[d]<x:
        return False
    while g<=d:
        m=(g+d)//2
        if t[m]==x:
            return True
        elif x<t[m]:
            d=m-1
        else:
            g=m+1
    return False
```

✔ La difficulté est d'estimer le nombre d'itérations. Écrivons $g = 2q + r$ et $d = 2q' + r'$ les divisions euclidiennes de g et d par deux. Ainsi $m = q + q' + r''$ où r'' est le reste de $r + r'$ par deux. On a donc

$$\begin{cases} m - 1 - g = q' - q + r'' - 1 - r \leqslant q' - q \\ d - m - 1 = q' - q + r' - r'' - 1 \leqslant q' - q \end{cases}$$

Ainsi le nombre de chiffres en base deux du contenu de `d-g` diminue d'au moins une unité à chaque itération. Le nombre d'itérations est donc un $O(c_n)$ où c_n est le nombre de chiffres en base deux de $n-1$, valeur initiale de `d-g`. On sait que $c_n = O(\ln n)$ (cf. info 1).

✔ Le nombre d'itérations est donc en $O(\ln n)$ et elles sont de coût constant : la complexité totale est en $O(\ln n)$.

✕ Le tri par sélection.

```
def tri_select(t):
    for i in range(len(t)):
        indmin=i
        for j in range(i+1,len(t)):
            if t[j]<t[indmin]:
                indmin=j
        t[i],t[indmin]=t[indmin],t[i]
```

✔ L'initialisation est de coût constant.

✔ La boucle sur l'indice i compote n itérations. L'intération d'indice i contient une initialisation en $O(1)$ puis une boucle sur l'indice j qui comporte $n-i-1$ itérations de coût constant.

✔ Comme

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

le coût total est en $O\left(n^2\right)$.

✕ Le tri par insertion.

```
def tri_insert(t):
    for i in range(1,len(t)):
        j=i
        while j>0 and t[j]<t[j-1]:
            t[j],t[j-1]=t[j-1],t[j]
            j=j-1
```

- ✔ L'initialisation est de coût constant.
- ✔ La boucle sur l'indice i compote $n-1$ itérations. L'intération d'indice i contient une initialisation en $O(1)$ puis une boucle sur l'indice j qui comporte i itérations (au maximum) de coût constant.
- ✔ Comme
$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O\left(n^2\right)$$
le coût total est en $O\left(n^2\right)$.
- ✔ On peut remarquer que la complexité est en $O(n)$ dans le meilleur des cas (celui d'une liste déjà triée pour laquelle la boucle d'insertion est de coût constant).

✘ Voici – sans justification – les complexités des algorithmes de tris que nous avons programmés :

Algorithme	**Pire des cas**	**Meilleur des cas**	**Coût spacial**
Tri par sélection	n^2	n^2	1
Tri par insertion	n^2	n	1
Tri rapide	n^2	$n\ln n$	n (au pire)
Tri par fusion	$n\ln n$	$n\ln n$	n

On peut aussi mentionner que tout algorithme de tri par comparaison admet une complexité au moins en $\Theta(n\ln n)$ dans le pire des cas. En dehors de cas particuliers (cf. le tri par dénombrement), on peut donc estimer que $O(n\ln n)$ est la complexité minimale d'un tri.

4.4. Complexité d'une fonction récursive

La complexité d'une fonction récursive est souvent plus délicate à estimer que dans le cas d'un algorithme itératif. Le principe de calcul est de déduire du code une relation de récurrence vérifiée par la suite $(C_n)_{n\in\mathbb{N}}$ puis de la résoudre (ou simplement de la majorer).

Revenons à l'exemple de la suite de Fibonacci.

```
def fibo1(n):
    if n<=1:
        return n
    return fibo1(n-2)+fibo1(n-1)
```

Notons C_n la complexité de l'appel `fibo1(n)` pour $n\in\mathbb{N}$. On a $C_0 = C_1 = 1$ (on compte un seul test). Soit $n \geqslant 2$. On a
$$C_n = 4 + C_{n-1} + C_{n-2}$$
car on compte un test initial et trois opérations (dont $n-1$ et $n-2$).

On en déduit facilement que
$$C_n \sim \sqrt{5}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} \quad \text{Remarque : } \frac{1+\sqrt{5}}{2} \approx 1{,}618$$
comme nous l'avons souligné en TP, mieux vaut éviter les appels récursifs multiples afin d'éviter une complexité exponentielle.

Nous avons proposé en TP un moyen de conserver l'aspect récursif tout en évitant l'écueil d'un double appel récursif. Ici, $C_0 = 1$ et pour tout $n \geqslant 1$

$$C_n = 5 + C_{n-1}$$

car on compte un test, le calcul de $n-1$, deux affectations et le calcul final d'une somme.

```
def fibo2(n):
    if n==0:
        return [0,1]
    [u0,u1]=fibo2(n-1)
    return [u1,u0+u1]
```

Ainsi $C_n = 5n + 1 \sim 5n$. On retrouve la complexité linéaire de l'algorithme itératif exposé dans les paragraphes précédents.

4.5. Complexité spaciale

Définition 2.6. Complexité spaciale

On appelle complexité spaciale le nombre de positions mémoire nécessitées l'algorithme.

Cette complexité dépend également de la machine utilisée et des données traitées. Certains mécanismes rendent son évaluation difficile : en Python, le *garbage collector* supprime des objets inutiles au fur et à mesure de l'exécution d'un programme (par exemple, un « *range* » n'est pas toujours généré in extenso).

Convention pour les calculs de complexité

Le terme complexité désignera systématiquement la complexité temporelle.

5. Énoncés des tests

2.1.

Prouver que la boucle ci-dessous termine en considérant les valeurs de l'expression $2p+3c$.

```
c,p=0,5
while p>0:
    if c==0:
        p,c=p-2,1
    else:
        p,c=p+1,0
```

2.2.

Trouver un invariant de boucle pour la fonction suivante :

```
def puissancesde2(n):
    p,c=1,n
    while c>0:
        p,c=2*p,c-1
    return(p)
```

2.3.

Donner sans justification un invariant de l'algorithme de codage en base 2 :

```
def codBin(n):
    t=[]
    while n>0:
        t.append(n%2)
        n=n//2
    m=len(t)
    return [t[m-1-i] for i in range(m)]
```

2.4.

Démontrer la correction de l'algorithme de recherche du minimum d'un tableau.

2.5.

Calculer la complexité de la fonction `puissancesde2`.

6. Solutions

2.1.

On va démontrer que $2p+3c$ est un variant de boucle. Il est clair que p et c sont à valeurs entières. On notera c, c' et p, p' les valeurs des variables sur deux itérations consécutives.

- Initialement, $(c,p) = (0,5)$ et $(c',p') = (1,3)$: on a bien $2p'+3c' = 10 < 15 = 2p+3c$ et $2p+3c, 2p'+3c'$ sont dans $\mathbb{N}$.
- Supposons la propriété vraie à la fin d'une itération.
  - Si $c = 0$, alors
  $$2p'+3c' = 2p-1 < 2p+3c$$
  et comme $p > 1$ et $p \in \mathbb{N}$, on a $2p-1 \geqslant 1$ d'où $2p'+3c' \in \mathbb{N}$.
  - Si $c \not\emptyset$, alors $c \geqslant 1$ car $c \in \mathbb{N}$. Ainsi,
  $$2p'+3c' = 2p+1 < 2p+3c$$
  Et $2p'+3c' \in \mathbb{N}$.

On en déduit que la boucle termine.

2.2.

C'est l'examen des valeurs des variables au cours des premières itérations de la boucle qui permet de conjecturer un invariant.

- La propriété $p = 2^{n-c}$ en fin d'itération est un invariant de boucle de l'algorithme proposé.
  - Initialement, $(p,c) = (1,n)$ donc la propriété est vraie.
  - Supposons la propriété vraie à la fin d'une itération. À la fin de l'itération suivante, (p,c) devient $(p',c') = (2p, c-1)$ ainsi
  $$p' = 2p = 2 \times 2^{n-c} = 2^{n-(c-1)} = 2^{n-c'}$$

2.3.

Pour $n = \underline{a_{m-1}\cdots a_0}_2 \neq 0$, la propriété

$$\begin{cases} n = \underline{a_{m-1}\cdots a_i}_2 \\ \\ t = [a_{i-1}, \ldots, a_0] \end{cases}$$

à la fin de la i-ième itération est un invariant de boucle.

2.4.

Rappelons d'abord l'algorithme.

```
def min(t):
    ind,n=0,len(t)
    for i in range(n):
        if t[i]<t[ind]:
            i=ind
    return t[ind]
```

On démontre facilement que la propriété

`t[ind]` est le minimum du sous-tableau `[t[0],...,t[i]]` en fin d'itération

est un invariant de boucle. En fin de boucle, $i = n-1$ donc que m est le minimum de t.

2.5.

La variable c prend successivement les valeurs $n, n-1, \ldots, 1, 0$. On dénombre n itérations dans la boucle `while` et, à chaque itération, un test, deux opérations et deux affectations. Comme il y a deux affectations initiales, la complexité vaut

$$C_n = 2+5n = \Theta(n)$$

La complexité est donc linéaire.