Qualité algorithmique

3e année

IUT de Villetaneuse

Axel Bacher

2024

Table des matières

Chapitre 1

Complexité

1.1 Introduction

On attend d'un algorithme qu'il résolve de manière *efficace* le problème à résoudre, *quelles que soient les données à traiter*. On s'intéresse principalement à deux qualités d'un algorithme donné :

— sa correction : résout-il bien le problème donné ?
— son efficacité : en combien de temps et avec quelles ressources ?

En effet, on n'exige pas simplement d'un algorithme qu'il résolve un problème. On veut également qu'il soit *efficace*, c'est-à-dire :

— rapide (en termes de temps d'exécution),
— peu gourmand en ressources (mémoire utilisée, bande passante réseau, etc.).

La *théorie de la complexité* (algorithmique) vise à répondre à ces besoins. Elle permet :

— de classer les problèmes selon leur difficulté,
— de classer les algorithmes selon leur efficacité,
— de comparer les algorithmes résolvant un problème donné afin de faire un choix éclairé *sans* devoir les implémenter.

Dans ce cours, nous verrons :

— comment concevoir des algorithmes ;
— comment démontrer leur correction et analyser leur efficacité (nous nous focaliserons sur le temps d'exécution).

Nous implémenterons nos algorithmes en Python, mais les principes développés dans ce cours s'appliquent aussi bien à n'importe quel langage.

Enfin, nous introduirons deux outils permettant de contrôler la correction et l'efficacité d'une application :

— un *profiler*, `cProfile`,
— un *debugger*, `pdb`.

Ces deux outils font partie d'une installation standard de Python, mais là encore, de nombreux équivalents existent pour essentiellement tous les langages.

1.2 Temps d'exécution

Evaluation de la rapidité d'un algorithme

On ne mesure pas la durée en heures, minutes, secondes, ... En effet, cela nécessiterait d'implémenter les algorithmes que l'on veut comparer. De plus, ces mesures ne seraient pas pertinentes car le même algorithme sera plus rapide sur une machine plus puissante.

Au lieu de cela, on utilise des *unités de temps abstraites* proportionnelles au nombre d'opérations effectuées. Au besoin, on pourra ensuite adapter ces quantités en fonction de la machine sur laquelle l'algorithme s'exécute.

Calcul du temps d'exécution

- Chaque **instruction basique** (affectation d'une variable, comparaison, $+$, $-$, $*$, $/$, ...) consomme une unité de temps.
- Chaque **itération** d'une boucle rajoute le nombre d'unités de temps consommées dans le corps de cette boucle.
- Chaque **appel de fonction** rajoute le nombre d'unités de temps consommées dans cette fonction.
- Pour connaitre le nombre d'unités de temps consommées par l'algorithme, on additionne le nombre d'unités de temps consommées par chaque morceau de l'algorithme.

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

L'algorithme suivant calcule $n! = n * (n-1) * (n-2) * \cdots * 2 * 1$ (avec $0! = 1$) :

```
def factorielle(n):
    fact = 1                          initialisation : 1
    i = 2                             initialisation : 1
    while i <= n:                itérations : au plus n − 1
        fact = fact * i   multiplication + affectation : 2
        i = i + 1              addition + affectation : 2
    return fact                  renvoi d'une valeur : 1
```

Comme il y a aussi un test à chaque itération, le nombre total d'opérations est

$$1 + 1 + (n-1) * 5 + 1 + 1 = 5n - 1.$$

Remarque:
- Ce genre de calcul n'est pas tout à fait exact. En effet, on ne sait pas toujours combien de fois exactement on va effectuer une boucle.
- De même, on ne sait pas toujours si une instruction dans un bloc `if` ou `if`... `else` sera effectuée ou pas, et cela affecte la complexité.
- Heureusement, on dispose d'outils plus simples à manipuler et plus adaptés.

1.3 Complexité algorithmique

La *complexité* d'un algorithme est une mesure de sa performance *asymptoptique* dans le *pire cas* ou *en moyenne*.
- Que signifie *asymptotique* ? Cela signifie que l'on s'intéresse à des données très grandes car les petites valeurs ne sont pas assez informatives.
- Que signifie « dans le pire cas » ? Cela signifie que l'on s'intéresse à la performance de l'algorithme dans les situations où le problème prend le plus de temps à être résolu. On est alors sûr que l'algorithme ne prendra jamais plus de temps que ce qu'on a estimé.
- Que signifie « en moyenne » ? Cela signifie que l'on s'intéresse à la moyenne des performances de l'algorithme dans toutes les situations où l'on souhaite résoudre le problème. On a alors une bonne estimation du temps que prendra l'algorithme s'il est utilisé dans un grand nombre de situations différentes.

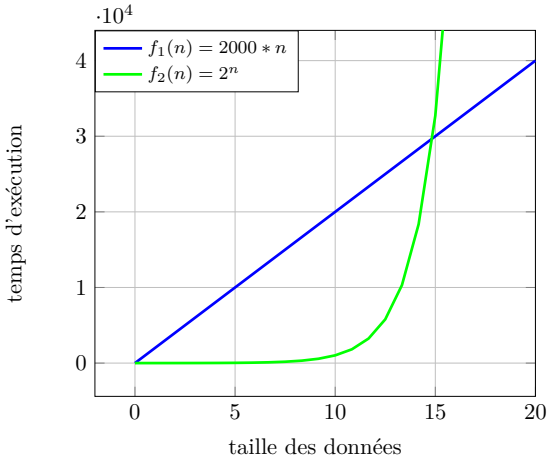
— Cela nécessite dans les deux cas de bien préciser quel est l'ensemble des situations envisagées.

Dans la suite du cours, on se focalisera sur la complexité algorithmique dans le pire cas.

Intuition : comportement asymptotique

Soit un problème à résoudre sur des données de taille n, et deux algorithmes résolvant ce problème en un temps $f_1(n)$ et $f_2(n)$, respectivement ;

Quel algorithme préférez-vous ?



La courbe de f_2 semble correspondre à un algorithme beaucoup plus efficace que celle de f_1... mais seulement pour de très petites valeurs ! Pour être plus précis, la complexité de f_1 est linéaire tandis que celle de f_2 est exponentielle.

1.4 La notation $O(\cdot)$

Les calculs à effectuer pour évaluer le temps d'exécution d'un algorithme peuvent parfois être longs et pénibles. De plus, le degré de précision qu'ils requièrent est souvent inutile. On aura donc recours à une *approximation* de ce temps de calcul, représentée par la notation $O(\cdot)$.

Notation

Soit n la taille des données à traiter, on dit qu'une fonction $f(n)$ est *en* $O(g(n))$ *(« en grand* O *de* $g(n)$ *»)* si :

$$\exists\, n_0 \in \mathbb{N}, \exists\, c \in \mathbb{R}, \forall\, n \geq n_0 : \quad |f(n)| \leq c|g(n)|.$$

Autrement dit : $f(n)$ est en $O(g(n))$ s'il existe un seuil à partir duquel la fonction $f(\cdot)$ est toujours dominée par la fonction $g(\cdot)$, à une constante multiplicative fixée près.

Ainsi dans l'exemple précédent $f_1(n) = O(f_2(n))$.

Exemples d'utilisation de $O(\cdot)$

Prouvons que la fonction $f_1(n) = 5n + 37$ est en $O(n)$:

— But : trouver une constante $c \in \mathbb{R}$ et un seuil $n_0 \in \mathbb{N}$ à partir duquel $|f_1(n)| \leq c|n|$.

— On remarque que $|5n + 37| \leq |6n|$ si $n \geq 37$:

$$|5 * 37 + 37| \leq 6 * |37|$$
$$|5 * 38 + 37| \leq 6 * |38|$$
$$\vdots$$

— On en déduit donc que $c = 6$ fonctionne à partir du seuil $n_0 = 37$, et on a fini.

Remarque: On ne demande pas d'optimisation (le plus petit c ou n_0 qui fonctionne), juste de donner des valeurs qui fonctionnent. Les valeurs $c = 10$ et $n_0 = 8$ sont donc aussi acceptables.

Prouvons que la fonction $f_2(n) = 6n^2 + 2n - 8$ est en $O(n^2)$:

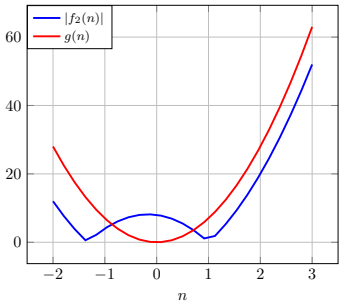
— On cherche d'abord la constante c, $c = 6$ ne peut pas marcher, on essaie donc $c = 7$.

— On doit alors trouver un seuil $n_0 \in \mathbb{N}$ à partir duquel :

$$|6n^2 + 2n - 8| \leq 7|n^2| \qquad \forall\ n \geq n_0.$$

— Un simple calcul nous donne $n_1 = -\frac{4}{3}$ et $n_2 = 1$ comme racines de l'équation $6n^2+2n-8 = 0$.

— En conclusion, $c = 7$ et $n_0 = 1$ nous donnent le résultat voulu.



Règles de calcul : simplifications

On calcule le temps d'exécution comme précédemment, mais on effectue les simplifications suivantes :

1. on oublie les constantes multiplicatives (elles valent 1),
2. on annule les constantes additives,
3. et on ne retient que les termes dominants.

Exemple: **simplifications**
Soit un algorithme effectuant $g(n) = 4n^3 - 5n^2 + 2n + 3$ opérations,

1. on remplace les constantes multiplicatives par 1 : $n^3 - n^2 + n + 3$,
2. on annule les constantes additives : $n^3 - n^2 + n$,
3. on garde le terme de plus haut degré : n^3,

et on a donc $g(n) = O(n^3)$.

Justification des simplifications

Les processeurs actuels effectuent plusieurs milliards d'opérations à la seconde.

1. Qu'une affectation requière 2 ou 4 unités de temps ne change pas l'ordre de grandeur de la complexité, d'où le remplacement des constantes par des 1 pour les multiplications.
2. Un nombre constant d'instructions est donc aussi négligeable par rapport à la croissance de la taille des données, d'où l'annulation des constantes additives.
3. Pour de grandes valeurs de n, le terme de plus haut degré l'emportera, d'où l'annulation des termes inférieurs.

On préfére souvent avoir une *idée* du temps d'exécution de l'algorithme plutôt qu'une expression plus précise mais inutilement compliquée, même si la connaissance de la constante multiplicative dominante fournit une information significative.

Règles de calculs : combinaison des complexités

— Les instructions de base prennent un temps constant, noté $\mathbf{O(1)}$.
— On additionne les complexités d'opérations en séquence :

$$O(f_1(n)) + O(f_2(n)) = O(f_1(n) + f_2(n))$$

— Même chose pour les branchements conditionnels :

Exemple:

```
if <condition>:          O(g(n))
    # instructions (1)   O(f1(n))
else:
    # instructions (2)   O(f2(n))
```

$\Big\} = O(g(n) + f_1(n) + f_2(n))$

Règles de calculs : combinaison des complexités

— Dans les boucles, on multiplie la complexité du corps de la boucle par le nombre d'itérations.
— La complexité d'une boucle `while` se calcule comme suit :

Exemple:

```
# en supposant qu'on a m itérations
while <condition>:                O(g(n))
    # instructions                O(f(n))
```

$\Big\} = O(m * (g(n) + f(n)))$

Calcul de la complexité d'un algorithme

— Pour calculer la complexité d'un algorithme :

1. on calcule la complexité de chaque « partie » de l'algorithme,
2. on combine ces complexités conformément aux règles qu'on vient de voir,
3. on simplifie le résultat grâce aux règles de simplifications qu'on a vues :
   — élimination des constantes, et
   — conservation du (des) terme(s) dominant(s).

Exemple: **factorielle de** n

Reprenons le calcul de la factorielle, qui nécessitait $5n - 1$ opérations.

```
def factorielle(n) :
    fact = 1                initialisation : O(1)
    i = 2                   initialisation : O(1)
    while i <= n :        n − 1 itérations : O(n)
        fact = fact * i      multiplication : O(1)
        i = i + 1           incrémentation : O(1)
    return fact                     renvoi : O(1)
```

Complexité de la procédure :

$$O(1) + O(n) \cdot O(1) + O(1) = O(n).$$

1.5 Classification(s)

Equivalence de fonctions en termes de $O(\cdot)$

$f(n) = O(g(n))$ n'implique pas $g(n) = O(f(n))$.
contre-exemple : $5n + 43 = O(n^2)$, mais $n^2 \neq O(n)$.
$f(n) \neq O(g(n))$ n'implique pas $g(n) \neq O(f(n))$.
contre-exemple : $18n^3 - 35n \neq O(n)$, mais $n = O(n^3)$.

On dit que deux fonctions $f(n)$ et $g(n)$ sont *équivalentes* si

$$f(n) = O(g(n)) \text{ et } g(n) = O(f(n)).$$

D'un point de vue algorithmique, trouver un nouvel algorithme de même complexité pour un problème donné ne présente donc pas beaucoup d'intérêt.

Quelques classes de complexité

On peut donc « ranger » les fonctions équivalentes dans la même *classe*. Voici quelques classes fréquentes de complexité (par ordre croissant en termes de $O(\cdot)$) :

Complexité	**Classe**
$O(1)$	constant
$O(\log n)$	logarithmique
$O(n)$	linéaire
$O(n \log n)$	quasi-linéaire
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel
$O(e^n)$	exponentiel

Hiérarchie

Pour faire un choix éclairé entre plusieurs algorithmes, il faut être capable de situer leur complexité. On fait une première distinction entre les deux classes suivantes :

1. les algorithmes dits *polynomiaux*, dont la complexité est en $O(n^k)$ pour un certain k,
2. les algorithmes dont la complexité ne peut pas être majorée par une fonction polynomiale (par exemple, les algorithmes *exponentiels*).

De même, un problème de complexité polynomiale est considéré « facile », sinon (complexité non-polynomiale ou inconnue (!)) il est considéré « difficile ».

Mais quel est l'intérêt de la classification des problèmes ?

« Recyclage » de résultats

On peut parfois montrer que deux problèmes donnés :
— sont équivalents,
— ou que l'on peut résoudre l'un à l'aide de l'autre.

Conséquences :
— Si on peut résoudre le problème P_1 en résolvant le problème P_2, alors on a directement un algorithme pour P_1 grâce à P_2,
— on a donc complexité$(P_1) \leq$ complexité(P_2)
— et on ne peut donc pas résoudre P_2 plus vite que P_1.

Exemple d'application : cryptosystème de Rabin

Inventé en 1979 par Michael O. Rabin, ce cryptosystème s'appuie sur le problème de factorisation, qu'on suppose difficile. Il s'agit d'un cryptosystème *asymétrique* : on chiffre avec une *clé publique* et on déchiffre à l'aide d'une *clé privée*.

La génération des clés fonctionne de la manière suivante :
— on choisit deux nombres premiers p et q, qui forment la clé privée ;
— la clé publique est le nombre $n = p * q$.

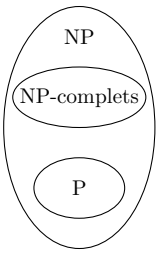
On chiffre les entrées en prenant leur carré modulo n. On déchiffre le message en calculant les racines carrées modulo p et q.

Si on connaît la clé privée, le déchiffrement est facile. Si on ne la connaît pas, il faut factoriser n.... mais on ne connaît pas d'algorithme de complexité polynomiale en b (le nombre de bits de n) pour factoriser un entier naturel n.

1.6 P, NP, $

Une question à un million de dollars

— *La classe P* est l'ensemble des problèmes qu'on peut résoudre avec un algorithme de complexité polynomiale.
— *La classe NP* est l'ensemble des problèmes dont on peut *vérifier* une solution avec un algorithme de complexité polynomiale.
— *La classe NP-complet* est l'ensemble des problèmes NP qui sont au moins aussi difficiles que tous les autres problèmes de la classe NP.



La question la plus importante en informatique théorique est :

$$\mathrm{P} \stackrel{?}{=} \mathrm{NP}$$

— Autrement dit : tout problème dont la solution est vérifiable en temps polynomial est-il également soluble en temps polynomial ?
— C'est aussi la plus lucrative : l'institut Clay offre $ 1.000.000 pour sa résolution.

Quelques problèmes faciles

— Rechercher un/le plus petit/le plus grand élément dans une base de données.
— Trier une base de données.
— Rechercher les occurrences d'un motif dans un texte.
— Calculer l'itinéraire le plus court entre deux sommets d'un graphe.

Quelques problèmes difficiles (NP-complets)

Problème: SUBSET SUM
Données : un ensemble S de n entiers.
Question : existe-t-il un sous-ensemble $S' \subseteq S$ dont la somme des éléments est nulle ?
Exemple : $S = \{-7, -3, -2, 5, 8\}$; $S' = \{-3, -2, 5\}$ est une solution.

Problème: PARTITION
Données : un ensemble S de nombres (répétitions autorisées).
Question : peut-on séparer S en deux sous-ensembles A et B tels que $\sum_{a \in A} a = \sum_{b \in B} b$?
Si $S = \{1, 3, 1, 2, 2, 1\}$, alors $A = \{1, 3, 1\}$ et $B = \{1, 2, 2\}$ marche.

Un problème de complexité inconnue

Problème: ISOMORPHISME DE GRAPHES
Données : deux graphes G_1 et G_2.
Question : G_1 et G_2 sont-ils "les mêmes" ? (peut-on numéroter leurs sommets de manière à obtenir deux ensembles d'arêtes identiques ?)

1.7 Mesures du temps de calcul et profilage

En complément de l'analyse théorique de la complexité d'un algorithme, on peut aussi contrôler l'efficacité pratique d'une implémentation donnée sur une machine. Pour mesurer le temps de calcul d'une fonction en Python, on peut utiliser le module `timeit`, comme suit :

```
1  def fact(n):
2      r = 1
3      i = 1
4      while i <= n:
5          r *= i
6          i += 1
7      return r
8
9  import timeit
10 timeit.timeit(lambda: fact(100000), number=1)
```

La fonction `timeit.timeit()` prend en premier argument l'expression à chronométrer (on peut préfixer `lambda:` pour accéder aux fonctions et variables qu'on a définies par ailleurs). L'argument `number=1` indique le nombre de fois que cette expression sera exécutée, ce qui est utile si le temps d'exécution est trop court pour être mesuré. Ici, cette fonction renvoie `2.297023976003402`, le temps en secondes que la machine a pris pour calculer 100000!.

Utilisation d'un profiler

Pour mesurer l'efficacité d'une application plus complexe, il devient utile d'utiliser un *profiler*, qui est capable de mesurer les performances *fonction par fonction* (certains profilers plus élaborés sont capables de mesurer *ligne par ligne*, utile pour comprendre quelles boucles prennent le plus de temps).

Ceci permet d'identifier les fonctions qui consomment le plus de ressources et ainsi de cibler celles qu'il faut optimiser. En effet, si une fonction prend quelques pourcents du temps d'exécution global, il est probablement peu utile de l'optimiser (au moins dans un premier temps). Si elle en prend la majorité, cela devrait au contraire être prioritaire.

De nombreux profilers, aux fonctionnalités variées, existent pour essentiellement tous les langages de programmation : `gprof` pour C, `VisualVM` pour Java, etc.

Exemple d'utilisation de `cProfile`

Une installation standard de Python fournit le profiler `cProfile`. En voici un exemple d'utilisation sur une implémentation du tri par tas :

```
1  def ajoute(l, n):
2      x = l[n]
3      p = (n-1) // 2
4      while n > 0 and x > l[p]:
5          l[n] = l[p]
6          l[p] = x
7          n = p
8          p = (n-1) // 2
```

```
9
10 def extrait_max(l, n):
11     x = l[n]
12     l[n] = l[0]
13     m = 0
14     while (2*m+1 < n and x < l[2*m+1]) or (2*m+2 < n and x < l[2*m+2]):
15         if 2*m + 2 < n and l[2*m+2] > l[2*m+1]:
16             l[m] = l[2*m+2]
17             m = 2*m + 2
18         else:
19             l[m] = l[2*m+1]
20             m = 2*m + 1
21     l[m] = x
22
23 def tri_par_tas(l):
24     for n in range(len(l)):
25         ajoute(l, n)
26     for n in range(len(l)-1, -1, -1):
27         extrait_max(l, n)
28
29 import random
30 import cProfile
31
32 l = []
33 for _ in range(100000):
34     l.append(random.random()) # liste de 100000 nombres aléatoires
35
36 cProfile.run('tri_par_tas(l)') # on trie la liste dans cProfile
37
38 for i in range(len(l) - 1):
39     assert l[i] <= l[i+1] # on vérifie que la liste est triée
```

Ceci produit la sortie suivante :

```
1          200006 function calls in 1.194 seconds
2
3    Ordered by: standard name
4
5    ncalls  tottime  percall  cumtime  percall filename:lineno(function)
6         1    0.000    0.000    1.194    1.194 <string>:1(<module>)
7    100000    0.057    0.000    0.057    0.000 tas.py:1(ajoute)
8    100000    1.089    0.000    1.089    0.000 tas.py:10(extrait_max)
9         1    0.048    0.048    1.194    1.194 tas.py:23(tri_par_tas)
10        1    0.000    0.000    1.194    1.194 {built-in method builtins.exec}
11        2    0.000    0.000    0.000    0.000 {built-in method builtins.len}
12        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Le profiler nous apprend aux lignes 7–8 que les fonctions `ajoute()` et `extrait_max()` ont été appelées 100000 fois chacune, ce qui est bien ce qui était attendu. De plus, les appels à `ajoute()` ont pris 57ms en tout, tandis que ceux à `extrait_max()` ont pris 1,089s, soit environ 91% du total (les deux fonctions ont la même complexité théorique, $O(\log n)$).

TD1 : Complexité

Exercice 1 :

Donnez la complexité des algorithmes suivants (`n` est une variable entière qui a été initialisée) :

a)

```
s = 0
i = 0
while i <= n:
    s = s + 1
    i = i + 1
```

b)

```
s = 0
i = 0
while i <= n:
    j = 0
    while j <= n:
        s = s + 1
        j = j + 1
    i = i + 1
```

c)

```
s = 0
i = 0
while i <= n:
    j = 0
    while j <= n * n:
        s = s + 1
        j = j + 1
    i = i + 1
```

d)

```
s = 0
i = 0
while i <= n:
    j = 0
    while j <= i:
        s = s + 1
    i = i + 1
```

e)

```
s = 0
i = 0
while i <= n:
    j = 0
    while j <= i*i:
        k = 0
        while k <= j:
            s = s + 1
            k = k + 1
        j = j + 1
    i = i + 1
```

Que devient la complexité de l'algorithme *d)* si l'on rajoute `j = j + 1` dans le corps de la seconde boucle ?

Exercice 2 :

Calculez la complexité du code suivant (vous pouvez supposer que la fonction `print()` est en $O(1)$, et que la variable `n` a été initialisée) :

```
i = 1
m = n
while i <= n:
    j = 1
    while j <= m:
        print(j)
        j = j + 1
        i = i + 1
    i = i + 1
```

Exercice 3 :

Calculez la complexité dans le pire cas du code suivant, où `n` et `a` sont des entiers préalablement initialisés :

```
i = 1
while i <= n:
```

```
    print(i)
    if i == a:
        j = 1
        while j <= n:
            print(j)
            j = j + 1
    i = i + 1
```

Exercice 4 :

Calculez la complexité du code suivant :

```
# x, m, n sont des variables entières déjà initialisées
r = 0
i = 1
while i <= n:
    j = 1
    while j <= m:
        r = 2 * (i + j) + r
        j = j + 1
    k = 1
    while k <= x:
        r = r * k
        k = k + 1
    print(r)
    i = i + 1
```

Exercice 5 :

Les algorithmes suivants utilisent une boucle pour réduire la valeur d'un entier naturel n à 0 et compter, à l'aide d'une variable c, le nombre d'itérations effectuées (on suppose que la variable n a été déclarée et initialisée).

1. Donnez la complexité de chacun de ces algorithmes en fonction de n.
2. Classez ces algorithmes par ordre croissant de temps d'exécution, en fonction des complexités calculées.
3. Lequel de ces trois algorithmes préfèrerait-on en pratique ?
4. Comment se comportent ces algorithmes quand n est négatif ?

a)

```
c = 0
while n > 0:
    n = n - 1
    c = c + 1
```

b)

```
c = 0
while n > 0:
    n = n // 2
    c = c + 1
```

c)

```
c = 0
while n > 0:
    n = n // 3
    c = c + 1
```

Exercice 6 : Exponentiation

Les deux questions ci-dessous doivent être résolues sans l'opérateur `**` ni la fonction `pow()`.

1. Écrivez une fonction calculant a^n pour n naturel, et donnez sa complexité.
2. Pour faire mieux, on remarque qu'on peut transformer une expression de type $a^n \times b$ de la façon suivante :
   — si n est pair, alors $a^n \times b = (a^2)^{n/2} \times b$.
   — si n est impair, alors $a^n \times b = (a^2)^{(n-1)/2} \times ab$;
   En appliquant les transformations ci-dessus à partir de $a^n \times 1$, écrivez une fonction de complexité $O(\log n)$ calculant a^n.

TP1 : Complexité

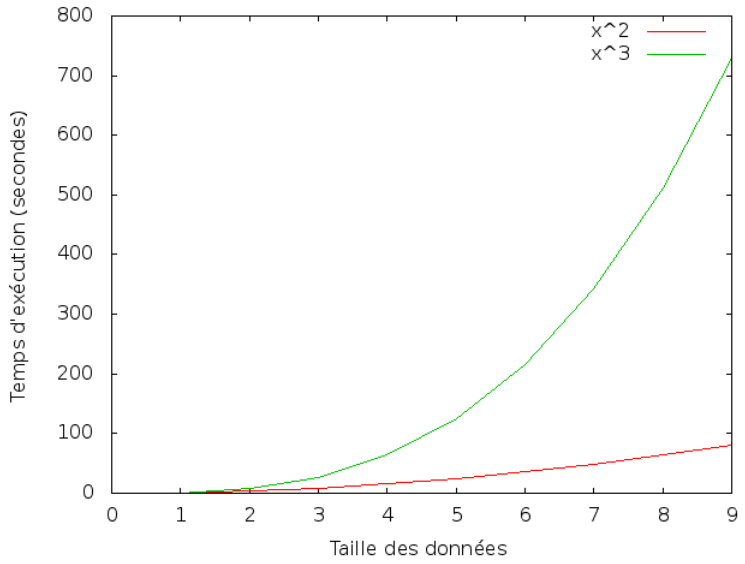
L'objectif de ce TP est de comparer les performances de divers algorithmes en théorie et en pratique dans des cas simples. Pour ce faire, outre les complexités, il faut également tester les programmes sur des données suffisamment grandes. Prenez garde cependant à commencer par de petites valeurs, pour détecter d'une part à partir d'où on commence à observer des différences, et pour éviter d'autre part de planter votre machine.

Récupérez à l'URL `http://lipn.fr/~bassino/complexite.py` le fichier `complexite.py`. Ce fichier contient une fonction `tracer(nom_fichier, *listes_et_titres)`, que vous utiliserez tout au long du TP. Cette fonction prend en paramètres un nom de fichier, des liste de données et des titres (**sans caractères spéciaux** (', ", ...)), et a pour effet de :

1. Tracer [1] un graphique comportant une courbe par jeu de données, en associant à chaque courbe le titre suivant les données ;
2. Sauver ce graphique sous la forme d'une image nommée `nom_fichier.png` dans le répertoire contenant `complexite.py`.

Le graphique ci-dessous, par exemple, a été produit à l'aide de la commande :

```
tracer("essai", [0, 1, 4, 9, 16, 25, 36, 49, 64, 81],       "x^2", \
                [0, 1, 8, 27, 64, 125, 216, 343, 512, 729], "x^3")
```



Exercice 7 : Exponentiation

1. Ecrivez un programme utilisant le module `timeit` et la fonction `tracer` pour produire un graphique comparant le temps d'exécution des deux fonctions d'exponentiation que vous avez définies dans l'exercice précédent. Il est recommandé de commencer les tests avec des valeurs de `n` relativement petites (50, 100, 200, ...), et de les augmenter progressivement.
2. Que peut-on dire des courbes résultantes ? En particulier, correspondent-elles bien à vos estimations théoriques ?

1. Cette fonction utilise le programme `gnuplot`(http ://www.gnuplot.info/), il faut donc veiller à l'installer si vous travaillez sur une autre machine que celles de l'IUT.

Exercice 8 : Micro-optimisations 1

Ecrivez un programme qui demande à l'utilisateur un entier `a` et un naturel `n` (ou "stop" pour arrêter le programme), et qui calcule $a \times 2^n$ de deux façons différentes :

1. avec une boucle ;
2. en effectuant un décalage de n bits vers la gauche. On peut décaler un entier `e` de `k` positions vers la gauche en Python avec l'instruction `e << k`.

Comparez les sommes des temps d'exécution sur `p` exécutions des deux méthodes pour des valeurs aléatoires de `a` et de `n`.

Exercice 9 : Micro-optimisations 2

La fonction `pow(a, b)`, qui calcule a^b, accepte un troisième argument facultatif : on peut utiliser `pow(a, b, c)` pour calculer le reste de la division de a^b par le naturel c. Ecrivez un programme comparant les temps d'exécutions des deux méthodes sur les données rentrées (`pow` contre les opérateurs de base) de la même manière que dans l'exercice précédent. Prenez une borne `M` suffisamment grande (~ 5000) pour la génération des entiers aléatoires.

Exercice 10 : Nombres premiers

Pour rappel, un nombre naturel est *premier* s'il possède exactement deux diviseurs, à savoir 1 et lui-même (donc 0 et 1 ne sont pas premiers, mais 2 oui). La page suivante : `http://primes.utm.edu/nthprime/` recense quelques-uns de ces nombres premiers.

1. Ecrivez une fonction `premier1(n)` renvoyant `True` si le naturel n donné est premier et `False` sinon, en cherchant un diviseur entre 2 et $n-1$.
2. On peut améliorer l'algorithme précédent en remarquant que le seul nombre premier pair est 2, et en ne cherchant donc que les diviseurs *impairs* de n. Implémentez cette amélioration dans une autre fonction `premier2(n)`.
3. Une autre amélioration, moins évidente, se base sur le fait que si un naturel n peut s'exprimer comme le produit des nombres a et b, alors $\min(a, b) \leq \sqrt{n}$ (en effet, si les deux facteurs étaient supérieurs à $\sqrt{n}$, alors leur produit serait supérieur à n).
   (a) Écrivez une nouvelle fonction `premier3(n)` intégrant cette amélioration à la fonction `premier1(n)` ;
   (b) Écrivez une nouvelle fonction `premier4(n)` intégrant cette amélioration à la fonction `premier2(n)` ;

   La fonction racine carrée en Python est `sqrt(n)`, accessible après avoir réalisé l'import `from math import sqrt`.
4. Complétez votre programme afin qu'il utilise la fonction `tracer` pour produire un graphique comparant les temps totaux d'exécution de vos fonctions, pour des valeurs croissantes du nombre `k` de bits dans les entiers générés. Le nombre `p` d'exécutions sera le même pour chaque valeur de `k`. Pour obtenir un entier aléatoire sur `k` (≥ 1) bits, on utilise la fonction `getrandbits(k)` du module `random`.

   Afin d'obtenir des graphiques interprétables, on vous recommande de ne pas dépasser 80 pour `p` et 16 pour `k`.
5. Constatez-vous des différences dans le temps d'exécution de ces fonctions ? Dans quels cas, et à partir de quelles valeurs ?
6. Donnez la complexité dans le pire cas de chacune des fonctions implémentées. Les résultats que vous obtenez en pratique vous semblent-ils cohérents par rapport à la théorie ?
7. Quel est le cas qui prendra le plus de temps à vérifier ?
8. Faites quelques tests en ne comparant que `premier3` et `premier4`. Obtenez-vous des différences frappantes ?

Chapitre 2

Récursivité

La *récursivité* est la propriété que possède un objet ou un concept de s'exprimer en fonction de lui-même. En algorithmique et en programmation, on dit qu'une fonction est *récursive* si elle s'appelle elle-même.

Il s'agit d'un concept important en algorithmique pour écrire des algorithmes, mais également pour décrire ou définir des structures de données. Cette technique rend l'écriture de code plus facile. Elle a son lot d'avantages, mais également d'inconvénients.

2.1 Fonctions récursives

Une fonction est récursive si son exécution peut conduire à sa propre invocation. Une telle fonction se présente donc comme suit :

```
# Fonction récursive
def f(P):    # P = liste de paramètres
    # instructions (1)
    x = f(Q) # appel avec d'autres paramètres
    # instructions (2)
    return resultat
```

Remarque : le renvoi n'est pas obligatoire.

Il faut toujours s'assurer que la fonction récursive qu'on écrit arrête à un moment de s'appeler. Une fonction récursive doit donc posséder une condition d'arrêt :

```
# Fonction récursive avec condition d'arrêt
def f(P):
    if test(P):
        return resultat1
    # instructions (1)
    # appel(s) récursif(s)
    # instructions (2)
    return resultat2
```

... et il faut bien sûr s'assurer qu'elle finira par être vraie. Par exemple :

```
def f(n): # n naturel
    if n == 0:
        print("!")
    else:
        print("*")
        f(n-1)
```

Le canevas d'une fonction récursive correcte se ramène au suivant :

```
# Fonction récursive correcte
def f(P): # P = paramètre(s)
    if test(P):
        # bloc sans appel récursif
    else:
        # bloc avec appel(s) récursif(s) utilisant
        # des paramètres "plus simples"
```

La notion de « plus simple » varie selon le contexte :
— si P est un naturel, alors $P' < P$;
— si P est une liste, alors P' est une liste plus petite.
Par exemple le calcul récursif de $n!$ conduit à

```
def factoR(n):
    if n <= 1:
         return 1
    else:
         return n*factoR(n-1)
```

et celui de a^n à

```
def puissR(a, n):
    if n == 0:
        return 1
    else:
        return a*puissR(a, n-1)
```

On a vu comment calculer la complexité des algorithmes traditionnels. Les règles ne changent pas dans le cas des algorithmes récursifs. Le nombre d'itérations de boucle est remplacé par le nombre d'appels récursifs.

Les variantes récursives de la factorielle ou de la puissance ne sont donc pas plus rapides en terme d'ordre de grandeur.

2.2 Fonctions récursives terminales

Une fonction est dite *récursive terminale* si elle ne fait aucun calcul après l'appel récursif. Par exemple, la fonction `puissR` ci-dessus n'est pas récursive terminale car elle effectue une multiplication par a après l'appel récursif. En revanche, la variante suivante l'est :

```
def puissRT(a, n, b=1): # calcule b * a^n
    if n == 0:
        return b
    else:
        return puissRT(a, n-1, b*a) # aucun calcul après l'appel récursif
```

Les fonctions récursives terminales sont intéressantes en pratique parce qu'elles permettent une optimisation évitant d'utiliser la pile d'exécution.

Dans la définition d'une fonction en Python, il est possible (et souvent souhaitable) de définir un argument par défaut pour chacun des paramètres. On obtient ainsi une fonction qui peut être appelée avec une partie seulement des arguments attendus.

Par exemple :

```
1 def politesse(nom, vedette ='Monsieur'):
2     print "Veuillez agréer ,", vedette, nom, ", mes salutations distinguées."
3
4 politesse('Dupont')
5 Veuillez agréer , Monsieur Dupont , mes salutations distinguées.
6
7 politesse('Durand', 'Mademoiselle')
8 Veuillez agréer , Mademoiselle Durand , mes salutations distinguées.
```

Lorsque l'on appelle cette fonction en ne lui fournissant que le premier argument, le second reçoit tout de même une valeur par défaut. Si l'on fournit les deux arguments, la valeur par défaut pour le deuxième est tout simplement ignorée.

On peut définir une valeur par défaut pour tous les paramètres, ou une partie d'entre eux seulement. Dans ce cas, cependant, les paramètres sans valeur par défaut doivent précéder les autres dans la liste.

TD2 : Récursivité

Exercice 1 : Nombres de Fibonacci

On rappelle que les nombres F_n de Fibonacci sont définis par

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad \text{si } n \geq 2. \end{cases}$$

1. Donnez une fonction récursive calculant ces nombres.
2. Utilisez `cProfile` pour compter le nombre d'appels récursifs pour calculer F_{30}. Qu'en pensez-vous ? Comment pourrait-on accélérer ce calcul ?

Exercice 2 :

Soit `L` une liste de nombres. Écrivez :

1. une fonction récursive renvoyant la somme des nombres de `L` (sans utiliser la fonction `sum`) ;
2. une fonction récursive renvoyant le produit des nombres de `L`. Si la liste est vide, renvoyez 0 dans les deux cas.

Exercice 3 :

Soit `L` une liste de nombres. Écrivez :

1. une fonction récursive renvoyant le minimum de `L` (sans utiliser la fonction `min`) ;
2. une fonction récursive renvoyant le maximum de `L` (sans utiliser la fonction `max`) ;
3. une fonction récursive renvoyant la position d'un élément donné dans `L` (sans utiliser la méthode `index`) ou `-1` si l'élément ne s'y trouve pas.
4. une fonction récursive renvoyant le nombre d'occurrences d'un élément donné dans `L` (sans utiliser la méthode `count`).

Pour les deux premières fonctions, si la liste passée en paramètre est vide, il faut afficher une erreur et renvoyer `None`.

Exercice 4 : Palindrome

Pour rappel, un *palindrome* est un mot qui se lit indifféremment de gauche à droite ou de droite à gauche (par exemple : « kayak » ou « radar »).

Écrivez une fonction récursive renvoyant `True` si la chaîne passée en paramètre est un palindrome, et `False` sinon (on suppose que tous les caractères sont des minuscules non accentuées).

Exercice 5 : Liste monotone

Écrivez une fonction récursive renvoyant `True` si une liste donnée est croissante, et `False` sinon. On renverra aussi `True` si la liste est vide ou si tous ses éléments sont les mêmes.

Exercice 6 : listes d'entiers monotones

Écrivez deux fonctions récursives qui renvoient une liste contenant les nombres de 1 à n. La première fonction renvoie les nombres dans l'ordre croissant, tandis que la deuxième les renvoie dans l'ordre décroissant (sans se servir de fonctions comme `reversed`).

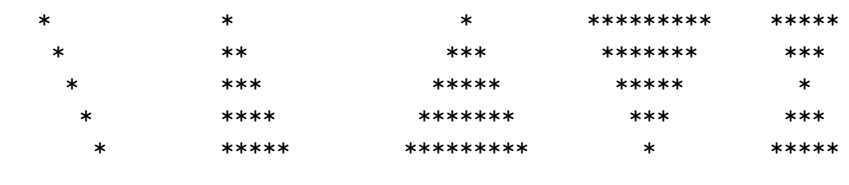
Exercice 7 : Tester la présence de doublons

Écrivez une fonction récursive renvoyant `True` si une liste donnée contient des doublons, et `False` sinon.

TP2 : Recursivité

Exercice 8 : Dessins récursifs

L'exercice consiste à écrire des fonctions récursives réalisant les dessins ci-dessous :



(a) (b) (c) (d) (e)

1. Ecrivez une fonction récursive `diagonale(n)` affichant une diagonale de `n` étoiles (voir (a)).
2. Ecrivez une fonction récursive `triangle(n)` affichant un triangle rectangle plein d'étoiles sur `n` lignes (voir (b)).
3. Ecrivez une fonction récursive `triangle2(n)` affichant un triangle isocèle plein d'étoiles sur `n` lignes (voir (c)).
4. Ecrivez une fonction récursive `triangle3(n)` affichant le même triangle que dans la question précédente, mais cette fois-ci sur sa pointe (voir (d)).
5. Ecrivez une fonction récursive `sablier(n)` affichant un sablier comme illustré en (e). Le sablier étant symétrique, la fonction affichera un sablier à `n` lignes si `n` est impair, et rien sinon.

Pour les questions 3, 4 et 5, il est nécessaire d'introduire un second paramètre.

Exercice 9 : `ls -R`

Le but de l'exercice est d'imiter l'action de la commande `ls -R` disponible sous les systèmes UNIX. Vous aurez besoin des fonctions suivantes du module `os` :

— `os.listdir(D)`, qui renvoie la liste de tous les fichiers se trouvant dans le répertoire `D` ;
— `os.path.isdir(F)`, qui renvoie `True` si le fichier `F` est un répertoire et `False` sinon ;
— `os.path.join(D, F)`, qui construit le chemin complet vers le fichier `F` dans le répertoire `D` (par exemple, si `F` est le fichier `truc.txt` situé dans le répertoire `/tmp/machin`, alors `os.path.join(D, F)` renverra `/tmp/machin/truc.txt`).

1. Ecrivez une fonction récursive `contenu(D)` affichant tous les fichiers dans un répertoire `D` donné. Cet affichage est récursif, c'est-à-dire que l'on affiche également le contenu des sous-répertoires de `D`, et de leurs sous-répertoires, et ainsi de suite.
2. On s'aperçoit que les affichages sont mélangés. Pour éviter cela, écrivez une autre fonction récursive `contenu2(D)`, qui cette fois affiche d'abord tous les éléments du répertoire courant avant de passer au contenu des sous-répertoires. Voici un exemple de répertoire sur le disque (gauche), avec ce que la fonction devra afficher (droite) :

```
/tmp/test:
    dir1:
        unfichierdansdir1.txt
        unautrefichierdansdir1.txt

    dir2:
        [ vide ]

    fichier.txt
    unautrefichier.txt
    unautrerep:
        fichier3.txt
        unfichier.txt
```

```
Contenu de /tmp/test:

dir2
unautrerep
dir1
unautrefichier.txt
fichier.txt

Contenu de /tmp/test/dir2:


Contenu de /tmp/test/unautrerep:

fichier3.txt
unfichier.txt

Contenu de /tmp/test/dir1:

unfichierdansdir1.txt
unautrefichierdansdir1.txt
```

3. Afin que l'affichage soit plus lisible, écrivez une fonction `contenu3(D)` faisant la même chose que `contenu2(D)`, mais affichant cette fois les éléments dans l'ordre alphabétique (toujours en distinguant les répertoires des fichiers). Vous pouvez utiliser la fonction `sorted()` dans votre correction. Dans le cas de l'exemple ci-dessus, la fonction affichera ceci (remarquez qu'on affiche d'abord, dans chaque répertoire, tous les répertoires, puis tous les fichiers) :

```
Contenu de /tmp/test:

dir1
dir2
unautrerep
fichier.txt
unautrefichier.txt

Contenu de /tmp/test/dir1:

unautrefichierdansdir1.txt
unfichierdansdir1.txt

Contenu de /tmp/test/dir2:


Contenu de /tmp/test/unautrerep:

fichier3.txt
unfichier.txt
```

Chapitre 3

Diviser pour régner

3.1 Présentation de la méthode

La méthode *diviser pour régner* est une méthode qui permet parfois de trouver des solutions efficaces à des problèmes algorithmiques. L'idée est de découper le problème initial, de taille n, en plusieurs sous-problèmes de taille sensiblement inférieure, puis de recombiner les solutions partielles.

3.1.1 Recherche dichotomique

Un premier problème basique consiste en la recherche d'un élément
Données : une liste `T`, un élément `x` ;
Question : est-ce que `T` contient `x` ? (si oui, donner sa position)
Un algorithme naïf consiste à examiner chaque position du tableau :

```
def recherche(T, x):
    for i in range(len(T)):      # examiner chaque position
        if T[i] == x:             # on a trouvé x
           return i
    return None
```

On renvoie la position si on a trouvé, `None` sinon.
La complexité de cet algorithme est simple à calculer :

```
def recherche(T, x):
    for i in range(len(T)):      # O(n)
        if T[i] == x:             # O(1)
           return i               # O(1)
    return None                   # O(1)
```

Cet algorithme est donc en $O(n)$ (pour une liste de n entiers).
Peut-on prouver qu'il est optimal, ou existe-t-il mieux ?

Recherche d'un élément dans une liste triée. Si on ne sait rien d'autre sur la liste, on ne peut pas faire mieux. Si en revanche `T` est triée, on peut aller beaucoup plus vite en procédant par *dichotomie* :

1. on examine l'élément du milieu, en position `p` ;
2. si `T[p]` vaut `x`, alors on a fini ;
3. sinon, on élimine une moitié de la liste et on poursuit notre recherche sur le reste.

Ce principe conduit à la fonction suivante :

```python
def dichotomie(T, x):
    a = 0
    b = len(T) - 1
    while a <= b:          # recherche entre les positions a et b
        p = (a + b) // 2   #  examiner l'élément du milieu
        if T[p] == x:      # on a trouvé x
            return p
        elif T[p] > x:     # éliminer les "supérieurs"
            b = p - 1
        else:              # éliminer les "inférieurs"
            a = p + 1
    return None            # None = "pas trouvé"
```

Analyse de complexité de la recherche dichotomique. On démarre avec un espace de recherche de n éléments. A chaque étape :

— on examine l'élément du milieu ; $O(1)$

— si ce n'est pas `x`, on resserre l'intervalle de recherche. $O(1)$

Quel est le pire cas ? Celui où `T` ne contient pas `x` et où on a dû aller « jusqu'au bout » pour s'en apercevoir. Dans le pire des cas, on s'arrête donc après que la recherche dans un intervalle de taille 1 a échoué. Comme le coût des opérations de la boucle est en $O(1)$, il suffit de déterminer le nombre maximal d'itérations effectuées.

Soit c le nombre maximal d'itérations de la boucle ; on a :

$$\frac{n}{\underbrace{2 \times 2 \times \cdots \times 2}_{c \text{ fois}}} = 1$$

Autrement dit, on a :

$$\begin{aligned} & 2^c = n \\ \Leftrightarrow \quad & \log_2 2^c = \log_2 n \\ \Leftrightarrow \quad & c = \log_2 n \end{aligned}$$

La complexité de la recherche dichotomique est donc $\mathcal{O}(\log n)$.

► Sur ce thème : **Exercice 1 du TD 2**

3.1.2 Exponentiation rapide

Il s'agit de calculer x^n pour x et n donnés, en calculant la complexité par rapport à n. La méthode naïve (multiplier n fois 1 par x) donne une complexité linéaire. On peut faire mieux en utilisant le fait que

$$\begin{cases} x^0 = 1, & \\ x^n = (x^2)^{\frac{n}{2}} & \text{si } n \text{ est pair et strictement positif,} \\ x^n = x * (x^2)^{\frac{n-1}{2}} & \text{si } n \text{ est impair.} \end{cases}$$

On peut directement traduire cette propriété en algorithme.

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

Le nombre d'appels récursifs est égal à $\log_2 n+1$ (la longueur de l'écriture binaire de n). Comme chaque appel a un coût constant, la complexité de l'exponentiation rapide est donc en $\mathcal{O}(\log n)$.

3.1.3 Tri fusion

Un exemple typique de la stratégie diviser pour régner est l'algorithme de tri fusion : pour trier un tableau `T` de taille n,

— on le découpe en deux tableaux de taille $n/2$

— et une étape de fusion permet de recombiner les deux solutions en $n-1$ opérations.

On peut l'écrire ainsi :

```
# fonction fusionnant deux listes triees
def fusion(T1,T2):
    if T1==[]:
        return T2
    if T2==[]:
        return T1
    if T1[0] < T2[0]:
        return [T1[0]]+fusion(T1[1:],T2)
    else:
        return [T2[0]]+fusion(T1,T2[1:])

# fonction triant recursivement les deux moities d'un tableau
# et fusionnant les ous-tableaux tries
def trifusion(T):
    if len(T) <=1:
        return T
    T1=T[:len(T)//2]
    T2=T[len(T)//2:]
    return fusion(trifusion(T1),trifusion(T2))
```

On va estimer la complexité en comptant le nombre $C(n)$ de comparaisons effectuées par l'algorithme. On a vu qu on obtient directement que

$$\begin{cases} C(0) = 0, \\ C(1) = 0, \\ C(n) \approx 2C(\frac{n}{2}) + n - 1. \end{cases}$$

Le symbole $\approx$ indique une approximation dans le calcul car il y a des parties entières à considérer pour être rigoureux.

On en déduit que la complexité de cet algorithme est en $\mathcal{O}(n \log n)$.

De plus, on peut montrer que ce tri est optimal dans le pire cas au sens où il faut toujours $\mathcal{O}(n \log n)$ comparaisons pour trier un tableau dans le pire cas.

▶ Sur ce thème : **Exercice 4 et 5 du TD 2**

3.2 Complexité des algorithmes diviser pour régner

La forme des algoritmes diviser pour régner considérés dans ce cours est :
- Diviser : on découpe le problème en a sous-problèmes de tailles n/b, qui sont de même nature, avec $a \geq 1$ et $b > 1$.
- Régner : les sous-problèmes sont résolus récursivement.
- Recombiner : on utilise les solutions aux sous-problèmes pour reconstruire la solution au problème initial en temps $\mathcal{O}(n^d)$, avec $d \geq 0$.

L'équation que l'on aura à résoudre une fois traduit le programme en équation sur la complexité est :

$$\begin{cases} T(1) = \text{constante}, \\ T(n) \approx aT\left(\frac{n}{b}\right) + \mathcal{O}(n^d). \end{cases}$$

Le théorème maitre permet de résoudre ce type d'equations.

Theorem 1 (Théorème maitre)**.** *On considère l'équation* $T(n) = aT(\frac{n}{b}) + \mathcal{O}(n^d)$. *Soit* $\lambda = \log_b a$. *On a les trois cas suivants :*

1. *si* $\lambda > d$, *alors* $T(n) = \mathcal{O}(n^\lambda)$;
2. *si* $\lambda = d$, *alors* $T(n) = \mathcal{O}(n^d \log n)$;
3. *si* $\lambda < d$, *alors* $T(n) = \mathcal{O}(n^d)$.

Par exemple, pour le tri fusion, on a $a = 2$, $b = 2$, $\lambda = d = 1$ et donc une complexité de $\mathcal{O}(n \log n)$.

En pratique, seuls les cas 1 et 2 peuvent mener à des solutions algorithmiques intéressantes. Dans le cas 3, tout le coût est concentré dans la phase « recombiner », ce qui signifie souvent qu'il y a des solutions plus efficaces.

Dans tous les exemples de ce chapitre on a approximé les formules pour ne pas faire apparaitre les parties entières et les légers décalages. Heureusement, Akra et Bazzi ont montré une extension du théorème maitre qui permet de travailler avec les approximations. On se reportera à Wikipedia pour en savoir plus sur ce sujet.

▶ Sur ce thème : **Exercice 2 et 3 du TD 2**

TD3 : Diviser pour régner

Exercice 1 : Ensemble d'entiers

Supposons que l'implémentation d'un ensemble S d'entiers positifs utilise les deux paramètres suivants :

1. un compteur N du nombre d'éléments dans S ;
2. un tableau $A[1 \dots M]$ dont les N premières valeurs sont les éléments de S dans l'ordre croissant.

M est choisi en sorte qu'il est toujours supérieur à N (pour les applications qui nous intéressent). Par exemple, pour $M = 8$ l'ensemble $\{2, 5, 7, 18\}$ sera implémenté par une paire de valeurs, dont la première est $N = 4$ et la deuxième est un tableau A de taille 8 :

$$< 4, [2, 5, 7, 18, -, -, -, -] > .$$

Montrer qu'on peut réaliser chacune des opérations suivantes en respectant les contraintes de temps (vous n'êtes pas obligés d'écrire les algorithmes en détail) :

a) MEMBRE (x, S) : recherche de l'élément x dans S en temps $O(\log|S|)$.
b) AJOUTER (x, S) : insertion de x à sa place dans le tableau en temps $O(|S|)$.
c) SUPPRIMER (x, S) : suppression de x et réduction du tableau en temps $O(|S|)$.
d) MIN (S) : recherche du plus petit élément de S en temps $O(1)$.
c) MAX (S) : recherche du plus grand élément de S en temps $O(1)$.

Exercice 2 : Théorème maitre

En utilisant le théorème maitre, retrouvez la complexité

1. de la recherche dichotomique,
2. de l'exponentiation rapide,
3. et du tri fusion.

Exercice 3 : Calcul de complexités à partir d'une relation de récurrence

1. En utilisant di nécessaire le théorème maitre, calculer la valeur asymptotique des complexités définies par les relations de récurrence.
   — $T_1(n) = 4 * T_1(n/2) + n^2$ et $T_1(0) = 0$.
   — $T_2(n) = 3 * T_2(n-1) + C$ où C est une constante et $T_2(0) = 0$.
   — $T_3(n) = 6 * T_3(n/4) + n^2$ et $T_3(0) = 0$.
   — $T_4(n) = 7 * T_4(n/5) + n$ et $T_4(0) = 0$.
   Attention : les divisions sont des divivsions entières ici $(3/2 = 1, 7/3 = 2, \dots)$
2. Ordonner des différentes complexités de la plus petite à la plus grande.

Exercice 4 : Tri fusion généralisé

Au lieu de couper un tableau en 2 parties et de trier chaque partie avant de recombiner les résulats, on pourrait augmenter le nombre de parts dans le tableau.

1. Ecrivez une généralisation du tri fusion où le tableau est coupé en trois parties
2. Quelle est la complexité théorique de cet algorithme ? Comment se compare-t-elle avec le tri fusion ordinaire ?

Exercice 5 : Recherche du minimum d'un tableau

1. Donner un **algorithme diviser pour régner** permettant de trouver le minimum d'un ensemble de n nombres stockés dans un tableau. Votre algorithme devra décrire précisément comment diviser le problème en sous-problèmes. Vous donnerez soit le pseudo-code, soit une description précise de l'algorithme.

   Attention : Seul un algorithme diviser pour régner sera accepté. On peut bien sûr trouver le miminum d'un tableau de taille n en temps $\mathcal{O}(n)$ en le parcourant.

2. Quelle est la complexité de cet algorithme diviser pour régner ? Qu'en pensez-vous ?

TP3 : Diviser pour régner

Exercice 6 : Recherche d'un élément

Le but de l'exercice est de comparer en pratique les performances de la recherche classique et de la recherche dichotomique. La génération de listes pouvant parfois être longue, on effectuera un certain nombre de recherches différentes sur la même liste, et on affichera le temps total d'exécution à l'utilisateur.

1. Ecrivez une fonction `listeAleatoireTriee(n)` qui utilise la fonction `random()` pour renvoyer une liste triée de n réels. Cette fonction doit être en $O(nf(\cdot))$, où $f(\cdot)$ est le temps d'exécution de la fonction `random()`.
2. implémentez les deux algorithmes de recherche vus au cours.
3. On veut à présent évaluer la croissance du temps moyen sur `p` exécutions du temps mis par les deux algorithmes en pratique. Complétez votre programme de manière à ce qu'il construise une liste triée aléatoire de `k` éléments pour chaque valeur de `k` entre 1 et `n`, et effectue `p` recherches d'un élément aléatoire dans cette liste avec chaque méthode. Vous pouvez choisir `n` et `p`, ou demander ces valeurs à l'utilisateur, mais évitez dans un premier temps de dépasser 1000 pour `n` et 200 pour `p`.
4. Afin de connaitre la croissance de ces fonctions, modifiez votre programme de sorte qu'il affiche (éventuellement sous forme graphique) la liste des temps **moyens** de recherche pour les deux méthodes (les moyennes sont donc calculées sur `p`) exécutions).
5. Au vu des résultats, serait-il intelligent de vérifier qu'une liste est triée avant de lancer une recherche ? Pourquoi ?
6. Python propose deux méthodes pour savoir si un élément est dans une liste : l'opérateur `in`, qui renvoie `True` si l'élément est dans la liste (utilisation : `x in T`), et la méthode `index()`, qui donne la position de `x` dans `T` si `x` s'y trouve (utilisation : `T.index(x)`).

   Comparez ces méthodes aux autres comme dans la question précédente, en produisant éventuellement un graphique contenant les quatre courbes.

   Attention : `T.index(x)` provoque une **exception** si `x` ne se trouve pas dans la liste `T`. Pour éviter les plantages, utilisez le bloc d'instructions suivant (les explications arriveront plus tard dans le cours de programmation) au lieu de `T.index(x)` :

```
essai = None
try:
    essai = T.index(x)
except: # x pas dans T
    pass
```

Exercice 7 : Nombre mystère

1. Ecrivez une fonction `mystere(n)` renvoyant le nombre de comparaisons nécessaires pour deviner un nombre aléatoire compris entre 0 et n. Plus précisément, le programme tire un nombre entre 0 et n puis essaie de deviner avec un minimum de comparaisons quelle est sa valeur. On commencera par afficher le nombre tiré aléatoirement et la valeur devinée par la fonction pour s'assurer de la bonne exécution du programme.
2. Complétez votre programme afin qu'il affiche (éventuellement sous forme graphique) le temps d'exécution de cette fonction pour des valeurs croissantes du nombre `k` de bits dans les entiers générés. Le nombre `p` d'exécutions sera le même pour chaque valeur de `k`. Pour obtenir un entier aléatoire sur `k` (≥ 1) bits, on utilise la fonction `getrandbits(k)` du module `random`.

Afin d'obtenir des résultats interprétables, on vous recommande de ne pas dépasser 80 pour `p` et 16 pour `k`.

3. Donnez la complexité dans le pire cas de cette fonction. Les résultats que vous obtenez en pratique vous semblent-ils cohérents par rapport à la théorie ?
4. Quel est le cas qui prendra le plus de temps à vérifier ? Quel est celui qui en prend le moins ?

Exercice 8 : Tri fusion

1. Implémenter le tri fusion.
2. Implémenter sa généralisation dans laquelle le tableau est partitionné en 3 parts.
3. Comparer expérimentalement la complexité de ces deux algorithmes. Qu'en pensez vous ? Cela correspond-il à leurs complexités théoriques relatives ?