Partie 4

Complexité

6 novembre 2018

Plan

1. Introduction
2. Approche empirique
3. Approche mathématique

Plan

1. Introduction
2. Approche empirique
3. Approche mathématique

Algorithmes et structures de données

- Vous avez maintenant toutes les bases de programmation en C pour pouvoir résoudre n'importe quel problème.
- Pour aborder un nouveau problème, vous ne devez pas partir de rien : Depuis plus de 50 ans, les informaticiens ont étudié et proposé des algorithmes et structures de données standards qui peuvent servir de briques de base pour aborder de nombreux problèmes.
- Dans la suite du cours, on verra les bases de ce domaine qui est un domaine scientifique à part entière [3]
- La résolution de problèmes informatiques requière systématiquement de combiner algorithmes et structures de données (cf. projet 1).
- Le critère de base pour l'analyse et l'évaluation de ces algorithmes et structures de données est leur performance (ou coût) en termes de temps de calcul et en termes d'espace mémoire consommé.

3. Ces notions seront largement approfondies dans le cours INFO0902 (et d'autres).

Pourquoi se soucier des performances ?

L'intérêt pratique d'un programme est très souvent dicté par ses performances :

- Est-il possible d'obtenir les résultats voulus en un temps raisonnable ?
- Peut-on traiter des volumes de données suffisamment importants ?

Étudier la performance d'un programme permet :

- de prédire son comportement
  - Est-ce que mon programme va se terminer ? Après combien de temps ?
- de comparer différents algorithmes et implémentations
  - Puis-je rendre mon programme plus rapide ? Si oui, comment ?

Améliorer les performances

L'amélioration des performances est la dernière étape du cycle de développement d'un programme.

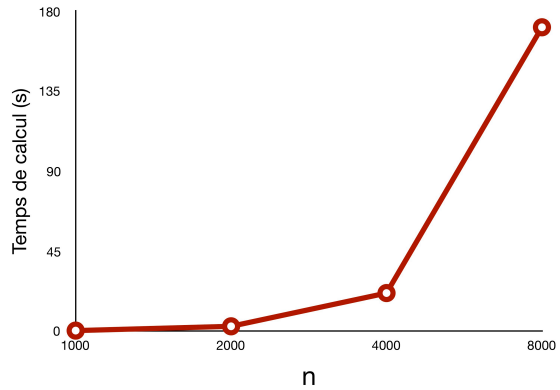
1. Ecriture du programme
2. Compilation $\Rightarrow$ si erreur (de syntaxe), retour en 1
3. Exécution et vérification du bon fonctionnement (tests unitaires) $\Rightarrow$ si erreur (de sémantique), retour en 1
4. Test du programme en situation réelle et sur de vraies données $\Rightarrow$ si erreur (de performance), retour en 1

Comme les autres étapes, on évite les itérations en y réfléchissant d'abord sur papier.

Analyse de performance

Pour analyser les performances d'un programme, on adopte une approche scientifique classique basée soit sur :

- l'expérimentation : on mesure les temps de calcul dans des conditions réelles
- la modélisation mathématique : on dérive une formule mathématique liant les performances aux données d'entrée.



$$T(n) = O(n^3)$$

Contrairement à d'autres sciences (chimie, biologie, physique, sociologie...), en sciences informatiques :

- les expérimentations sont quasi gratuites
- la modélisation mathématique est nettement plus aisée

Illustration : problème 3SUM

Le problème 3SUM :

> *Étant donné n nombres entiers, énumérer tous les triplets de valeurs sommant à 0.*

Trouve de nombreuses applications pratiques et théoriques.

Solution naïve (pour le comptage) : on énumère tous les triplets de valeurs et on teste leur somme.

```
int count_3sum(int tab[], int n) {
  int count = 0;
  for (int i = 0; i < n-2; i++)
    for (int j = i+1; j < n-1; j++)
      for (int k = j+1; k < n; k++)
        if (tab[i]+tab[j]+tab[k] == 0)
          count++;
  return count;
}
```

Combien de temps prendra ce programme pour un tableau d'un million de valeurs ?

Plan

1. Introduction

2. Approche empirique

3. Approche mathématique

Approche empirique

Principe : On implémente l'algorithme, on l'exécute et on mesure ses performances.

Il faut trouver des données représentatives sur lesquels tester l'algorithme. Deux options :

- On collecte des données réelles
- On écrit un programme pour générer des données

Exemple : un générateur de données pour le problème 3SUM

```
void generate_3sum_data(int tab[], int n, int m) {
  for (int i = 0; i < n; i++)
     tab[i] = rand() % (2*m) - m;
}
```

Génère un tableau aléatoire de n nombres entiers pris dans $\{-m, \ldots, m-1\}$. La probabilité de trouver des triplets sommant à zéro dépend de n et m.

Comment mesurer les temps de calcul ?

Pour mesurer le temps d'exécution d'un programme, on peut utiliser les fonctions de `time.h`

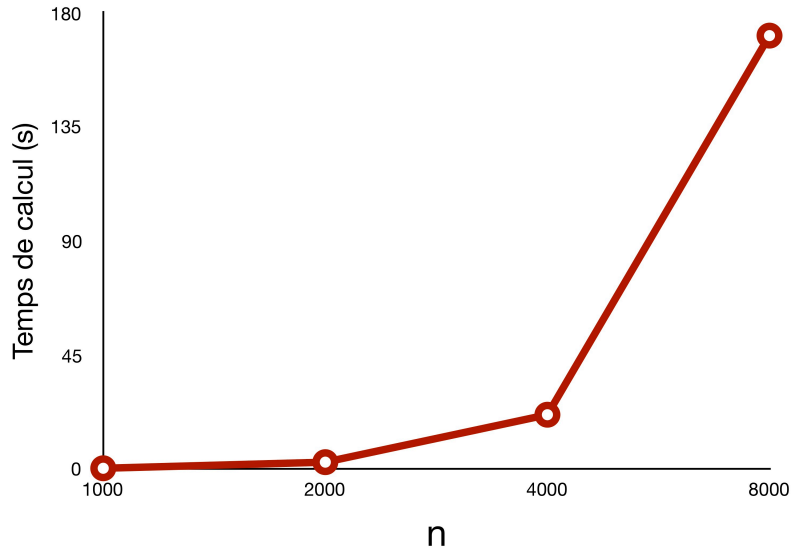
```
#include <time.h>

....
clock_t begin = clock();

// The code you want to monitor should be here

clock_t end = clock();

double time_spent = (double)(end-begin) / (double)CLOCKS_PER_SEC;
```

La variable `time_spent` contient le temps (en secondes) pris par le code entre les appels à `clock()`.

Illustration sur 3SUM

On double la taille du tableau d'un essai à l'autre avec des entiers compris entre -1000000 et +999999.

n	temps (s)
1000	0.34
2000	2.75
4000	21.45
8000	170.9



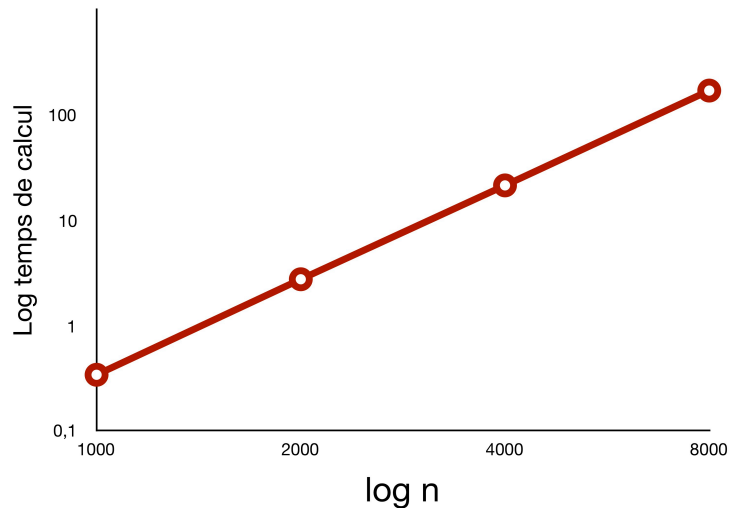
(Sur un Intel Core i7 2.5 GHz)

Analyse des données

Tracer la courbe sur une échelle logarithmique :

- Si les points sont sur une droite, la courbe est de la forme $a.n^b$ (c'est souvent le cas).
- L'exposant b est donné par la pente de la courbe.
- Le facteur a peut s'obtenir à partir des données

n	$T(n)$	$\log_2 n$	$\log_2 T(n)$	$T(n+1)/T(n)$
1000	0,34	10	-1,5	-
2000	2,75	11	1,5	8
4000	21,45	12	4,4	7,8
8000	170,9	13	7,4	8



$$\begin{aligned} T(n) &\approx an^b \Rightarrow b \approx 3 \text{ et } a \approx \frac{1}{3}10^{-9} \\ &\approx \frac{1}{3}10^{-9}n^3 \end{aligned}$$

Prediction et vérification

Hypothèse : le temps de calcul de `count_3sum` est $\approx \frac{1}{3}10^{-9}n^3$.

Vérification :

- Pour $n = 16000$, la fonction devrait prendre 1365 secondes.
- Temps observé : 1375 secondes (23 minutes).

Prédiction : Pour $n = 1$ million, la fonction prendra 333 millions de secondes ($>$ 10 ans).

Temps de calcul moyens

Les temps de calcul peuvent fluctuer fortement en fonction des données.

Exemple :

- vérification de l'occurrence d'un triplet sommant à zéro plutôt que comptage :

```
int has_3sum(int tab[], int n) {
  for (int i = 0; i < n-2; i++)
    for (int j = i+1; j < n-1; j++)
      for (int k = j+1; k < n; k++)
        if (tab[i] + tab[j] + tab[k] == 0)
          return 1;
  return 0;
}
```

- Pour $n = m = 1000000$, les temps de calcul peuvent aller de 0 (le premier triplet testé somme à 0) à plus de 10 ans (aucun triplet ne somme à zéro).

Dans ce cas, il est utile de répéter l'expérience plusieurs fois avec des données aléatoires et de calculer la moyenne (et l'écart-type) des temps de calcul.

Avantages et limitations de l'approche empirique

Avantages :

- Expériences très faciles à réaliser.
- On mesure les temps de calcul de l'implémentation réelle.

Limitations :

- Demande d'implémenter l'algorithme (pour peut-être se rendre compte qu'il est inefficace).
- Temps de calcul dépendent de l'implémentation et de la machine, même à solution algorithmique fixée.
- Ne fournit pas une preuve formelle de l'évolution des temps de calcul avec n. Tirer des conclusions à partir de valeurs expérimentales peut mener à des erreurs.

Plan

1. Introduction

2. Approche empirique

3. Approche mathématique
   - Illustration
   - Notation asymptotique
   - En pratique

Approche mathématique

Principe :

- On fait des hypothèses sur le modèle d'exécutions du programme
  - Exécution séquentielle (pas de parallélisme)
  - Les instructions élémentaires prennent une temps constant
  - ...
- On compte le nombre de fois que chaque instruction est exécutée.
- On obtient les temps de calcul en sommant le temps d'exécution (constant) de chaque instruction multiplié par son nombre d'exécutions.

Le temps dépend en général des données d'entrée :

- On exprime le temps de calcul en fonction de la taille des données d'entrée.
- Si les nombres d'exécutions des instructions dépendent de l'entrée, on se place dans le cas le plus défavorable (*worst case*).

Illustration : 2SUM

```
1  int count_2sum(int tab[], int n) {
2    int count = 0;
3    for (int i = 0; i < n-1; i++)
4      for (int j = i+1; j < n; j++)
5          if (tab[i] + tab[j] == 0)
6            count++;
7    return count;
8  }
```

Ligne	Coût	Nombre d'exécutions
2	c_1	1
3	c_2	$n-1$
4	c_2	$\sum_{i=0}^{n-2}(n-i)$
5	c_3	$\sum_{i=0}^{n-2}(n-i-1)$
6	c_4	n_4
7	c_5	1

Sachant que

- $\sum_{i=0}^{n-2}(n-i) = \frac{n(n+1)}{2} - 1$
- $\sum_{i=0}^{n-2}(n-i-1) = \frac{(n-1)n}{2}$ (le nombre de paires à tester)
- n_4 dépend du tableau mais vaut au pire cas $\frac{(n-1)n}{2}$ (toutes les paires somment à 0)

on a :

$$T(n) = \frac{c_2 + c_3 + c_4}{2} n^2 + \frac{3c_2 - c_3 - c_4}{2} n + (c_1 - 2c_2 + c_5)$$

Illustration : 2SUM

$$T(n) = an^2 + bn + c$$

Les constantes a, b, c et d dépendent :

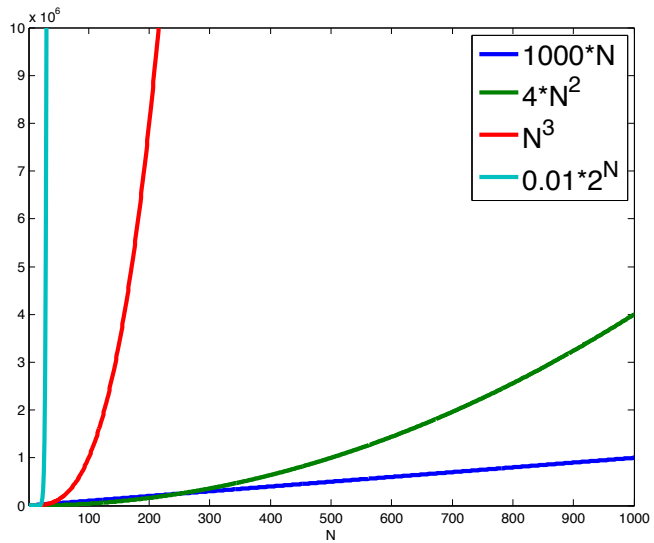
- de l'implémentation exacte (via les nombres d'exécutions)
- de la machine (via les constantes c_i)

Idéalement, on aimerait caractériser les performances d'un algorithme indépendamment de l'implémentation et de la machine.

Solution : on se focalise sur la vitesse de croissance asymptotique des temps de calcul.

Analyse asymptotique

- On s'intéresse à la vitesse de croissance ("order of growth") de $T(n)$ lorsque n est très grand ($n \to \infty$).
  - Tous les algorithmes sont rapides pour des petites valeurs de n
- On simplifie généralement $T(n)$:
  - en ne gardant que le terme dominant
    - Exemple : $T(n) = 10n^3 + n^2 + 40n + 800$
    - T(1000)=1000001040800, $10 \cdot 1000^3 = 10000000000$
  - en ignorant le coefficient du terme dominant
    - Asymptotiquement, ça n'affecte pas l'ordre relatif



- Exemple : 2SUM : $T(n) = an^2 + bn + c \to n^2$.

Pourquoi est-ce important ?

- Supposons qu'on puisse traiter une opération de base en $1\mu s$.
- Temps d'exécution pour différentes valeurs de n

T(n)	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
n	$10\mu s$	$0.1ms$	$1ms$	$10ms$
$400n$	$4ms$	$40ms$	$0.4s$	$4s$
$2n^2$	$200\mu s$	$20ms$	$2s$	$3.3m$
n^4	$10ms$	$100s$	$\sim$ 11.5 jours	317 années
2^n	$1ms$	4×10^{16} années	3.4×10^{287} années	...

(Dupont)

Pourquoi est-ce important ?

- Taille maximale du problème qu'on peut traiter en un temps donné :

T(n)	en 1 seconde	en 1 minute	en 1 heure
n	1×10^6	6×10^7	3.6×10^9
$400n$	2500	150000	9×10^6
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

- Si m est la taille maximale que l'on peut traiter en un temps donné, que devient cette valeur si on reçoit une machine 256 fois plus puissante ?

T(n)	Temps
n	$256m$
$400n$	$256m$
$2n^2$	$16m$
n^4	$4m$
2^n	$m + 8$

(Dupont)

Notation asymptotique "grand-O"

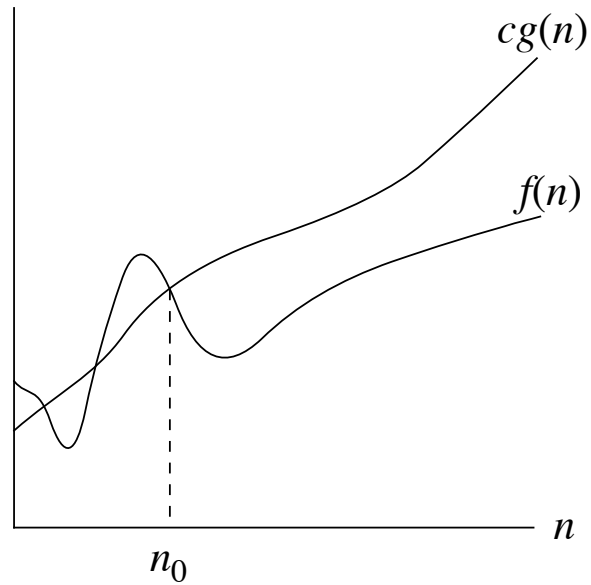
Les vitesses de croissance sont généralement précisées en utilisant la notation asymptotique "grand-O" (ou encore notation de Landau).

Définition : Soient f et g deux fonctions $\mathbb{N} \to \mathbb{R}^+$. On dira que

$$f \in O(g)$$

ssi

$$\exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ : \forall n > n_0 : f(n) \leq cg(n)$$



Par abus de notation, on écrira aussi : $f(n) \in O(g(n))$ ou $f(n) = O(g(n))$.

Complexité en temps

On dira qu'un algorithme a une complexité en temps $O(f(n))$ (ou plus simplement est $O(f(n))$) si ses temps de calcul dans le pire cas $g(n) \in O(f(n))$.

- Exemple : La complexité de `count_2sum` est $O(n^2)$.

Remarque importante : La notation grand-O sert à exprimer une borne supérieure sur la complexité.

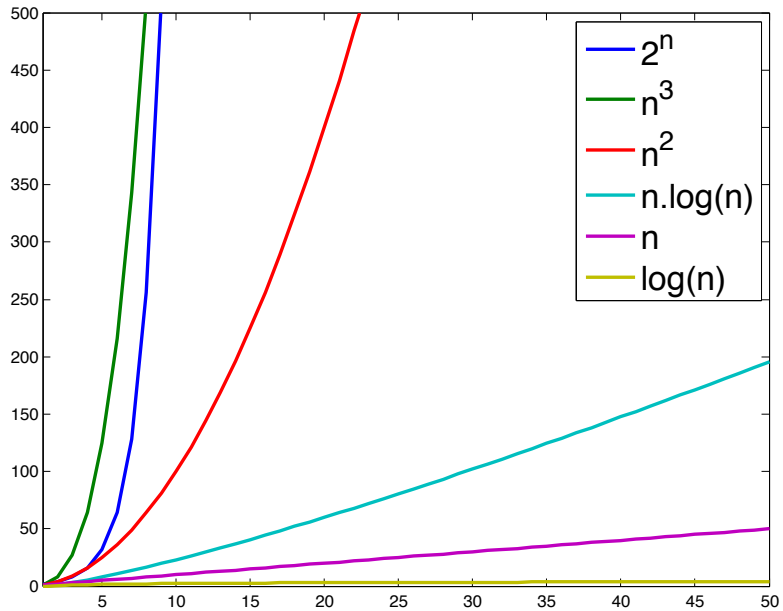
- Généralement, quand on dit qu'un algorithme est $O(f(n))$, on suppose que $O(f(n))$ est le plus petit sous-ensemble contenant la fonction $g(n)$ exprimant les temps de calcul de l'algorithme dans le pire cas.
- Par ex. : on ne dira pas que `count_2sum` est $O(n^3)$ même si $O(n^2) \subset O(n^3)$.

Exemples :

- $n^2 + 2n + 2 \Rightarrow O(n^2)$
- $n^2 + 100000n + 3^{1000} \Rightarrow O(n^2)$
- $\log(n) + n + 4 \Rightarrow O(n)$
- $10^{-4} n \log(n) + 3000n \Rightarrow O(n \log(n))$
- $2n^{30} + 3^n \Rightarrow O(3^n)$

Hiérarchie de classes de complexité

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^{a>1}) \subset O(2^n)$$



Classes de complexité : dénominations et exemples

Complexité constante	$O(1)$	Instructions élémentaires
Complexité logarithmique	$O(\log n)$	Recherche dichotomique (cf. partie 5)
Complexité linéaire	$O(n)$	Parcours d'un tableau 1D (par ex., recherche du maximum)
Complexité linéarithmique	$O(n \log n)$	Tri par fusion (cf. partie 5)
Complexité quadratique	$O(n^2)$	Parcours d'un tableau 2D, 2SUM
Complexité cubique	$O(n^3)$	Multiplication matricielle naïve
Complexité exponentielle	$O(2^n)$	Tour de Hanoï (partie 2), énumération des sous-ensembles d'un ensemble
Complexité factorielle	$O(n!)$	Approche naïve du voyageur de commerce

Analyse de complexité en pratique

En pratique, il n'est (la plupart du temps) pas nécessaire de compter explicitement le nombre d'exécutions de chaque instruction.

On peut calculer la complexité en notation grand-O directement en se basant sur les propriétés suivantes :

- Si $f(n) \in O(g(n))$, alors pour tout $k \in \mathbb{N}$, on a $k \cdot f(n) \in O(g(n))$
  - Exemple : $\log_a(n) \in O(\log_b(n))$, $a^{n+b} \in O(a^n)$

- Si $f_1(n) \in O(g_1(n))$ et $f_2(n) \in O(g_2(n))$, alors $f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$ et $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$
  - Exemple : $\sum_{i=1}^{m} a_i n^i \in O(n^m)$

- Si $f_1(n) \in O(g_1(n))$ et $f_2(n) \in O(g_2(n))$, alors $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$

Analyse de complexité en pratique

Quelques règles simples :

- Affectation, accès à un tableau, opérations arithmétiques, appel de fonction : $O(1)$
- Instruction If-Then-Else : O(complexité max des deux branches)
- Séquence d'opérations : l'opération la plus couteuse domine (règle de la somme)
- Boucle simple : $O(nf(n))$ si le corps de la boucle est $O(f(n))$

Analyse de complexité en pratique

- Double boucle complète : $O(n^2 f(n))$ où $f(n)$ est la complexité du corps de la boucle
- Boucles incrémentales : $O(n^2)$ (si corps $O(1)$)

```
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
      ...
```

- Boucles avec un incrément exponentiel : $O(\log n)$ (si corps $O(1)$)

```
for (int i = 1; i <= n; i = 2*i)
   ...
```

Exemple

But : calculer les moyennes des préfixes d'un tableau : $\texttt{prefix}[i] = \frac{\sum_{j=0}^{i} \texttt{tab}[j]}{i+1}$

```
float *prefixAverage(float tab[], float n) {
  float *prefix = malloc(n*sizeof(float));
  for (int i = 0; i < n; i++) {
    float a = 0;
    for (int j = 0; j <= i; j++)
      a += tab[j];
    prefix[i] = a/(i+1);
  return prefix;
}
```

$T(n) = O(n^2)$

```
void prefixAverage2(float tab[], float n) {
  float *prefix = malloc(n*sizeof(float));
  float s = 0;
  for (int i = 0; i < n; i++) {
    s += tab[i];
    prefix[i] = s/(i+1);
  }
  return prefix;
}
```

$T(n) = O(n)$

Exemple : 3sum

```
1  int count_3sum(int tab[], int n) {
2    int count = 0;
3    for (int i = 0; i < n-2; i++)
4      for (int j = i+1; j < n-1; j++)
5        for (int k = j+1; k < n; k++)
6          if (tab[i]+tab[j]+tab[k] == 0)
7            count++;
8    return count;
9  }
```

- Ligne 3 exécutée $O(n)$ fois.
- Ligne 4 exécutée $O(n^2)$ fois.
- Lignes 5, 6, et 7 exécutées une fois par triplet de valeurs. Nombre de triplets :

$$C_n^3 = \frac{n(n-1)(n-2)}{6} \in O(n^3).$$

$\Rightarrow$ Complexité en temps : $O(n^3)$.

Limitations de l'analyse asymptotique

- Les facteurs constants ont de l'importance pour des problèmes de petites tailles
  - Il vaut mieux un algorithme s'exécutant en $O(n^2)$ secondes qu'un algorithme s'exécutant en $O(\log n)$ années.
- Deux algorithmes de même complexité (grand-O) peuvent avoir des propriétés très différentes
  - Comme `count_3sum`, les deux algorithmes suivants sont $O(n^3)$
  - `count_3sum_2` est 6 fois plus lent et `has_3sum` peut traiter des tableaux aléatoires de taille 1 million en moins d'un centième de seconde.

```
int count_3sum_2(int tab[], int n) {
  int count = 0;
  for (int i = 0; i < n-2; i++)
    for (int j = 0; j < n-1; j++)
       for (int k = 0; k < n; k++)
          if (i < j && j < k)
             if (tab[i]+tab[j]+tab[k] == 0)
                count++;
  return count;
}
```

```
int has_3sum(int tab[], int n) {
  for (int i = 0; i < n-2; i++)
    for (int j = i+1; j < n-1; j++)
      for (int k = j+1; k < n; k++)
        if (tab[i]+tab[j]+tab[k] == 0)
          return 1;
  return 0;
}
```

$\Rightarrow$ Important de tester l'algorithme dans des conditions réelles (ou de réaliser une analyse mathématique plus fine).

Complexité d'algorithmes récursifs

L'analyse de la complexité d'algorithmes récursifs mène généralement à une équation récurrente, dont la résolution n'est pas toujours aisée.

Exemple :

- fonction factorielle :

```
int fact(int n) {
  if (n <= 1)
     return n;
  return n * fact(n-1);
}
```

$$\begin{aligned} T(0) &= c_0 \\ T(n) &= T(n-1) + c_1 \end{aligned}$$

- Solution : $T(n) = c_1 n + c_0 \in O(n)$

On se contentera de voir quelques cas particuliers dans ce cours.

Complexité en espace

La complexité en espace d'un algorithme mesure l'espace mémoire utilisé par l'algorithme en fonction de la taille de l'entrée.

Comme la complexité en temps :

- On la calcule dans le pire cas.
- On l'exprime en utilisant la notation grand-O.

Dans le cas des algorithmes récursifs, il faut prendre en compte l'espace mémoire nécessaire au stockage du contexte des appels récursifs, qui est proportionnel à la profondeur de l'arbre des appels récursifs.

Par exemple : la complexité en espace de `fact` est $O(n)$.

Exercice

Quelles sont les complexités en temps et en espace de la fonction `pow_rec2` ?

```
float pow_rec2(float a, int x) {
    if (x == 1)
       return a;
    if (x % 2 == 0) // x pair
       return pow_rec2(a * a, x/2);
    else            // x impair
       return a * pow_rec2(a * a, (x-1)/2);
}
```