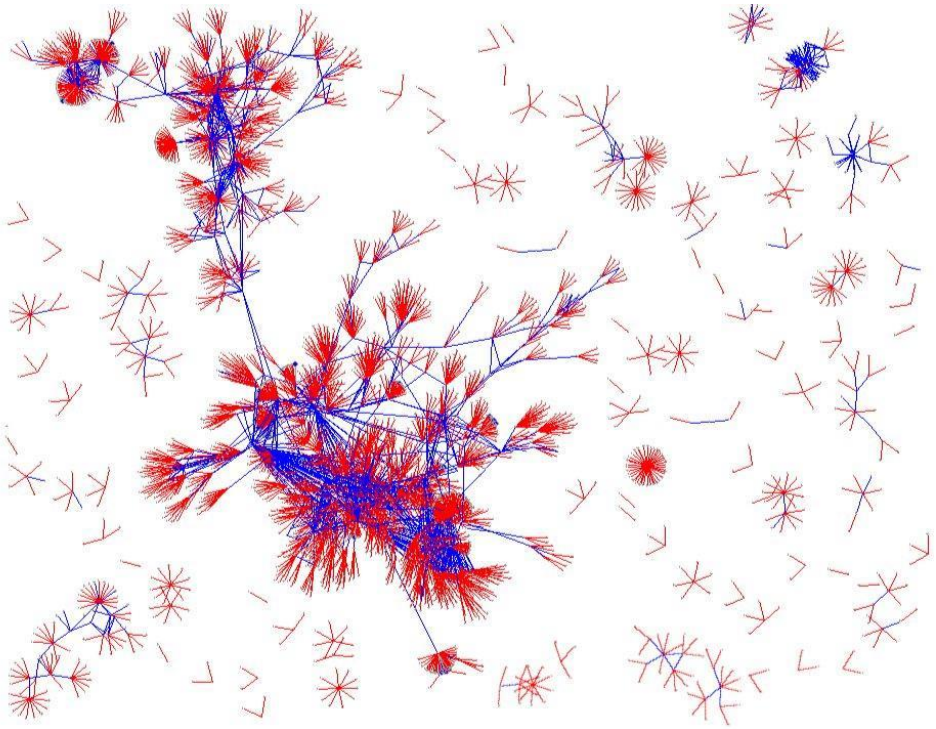


R5.A.04

Qualité Algorithmique

Chapitre1

Correction, complexité

Supports : Régis Fleurquin

Plan du cours et évaluation

- Plan du cours (indicatif)
  - (Séance 1) Qualité des algorithmes – Complexité temporelle
  - (Séance 2) Qualité (informatique) du logiciel – Normes - Métriques du logiciel
  - (Séance 3) Outils d'audit de la qualité du code
  - (Séance 4) Outils d'audit de la sécurité
  - (Séance 5) Tests de charge (performance et disponibilité)
  - (Séance 6) Chaînes de CI et qualité

- Évaluation via un oral (exposé/démo) prévu lors de la dernière séance (Séance 7) :
  - Mise en place d'outils d'évaluation de la qualité d'un précédent projet que vous avez réalisé via une chaîne de CI

1. Qualité théorique des algorithmes (et donc des programmes)

- On vérifie sur les algorithmes deux propriétés fondamentales :
  - **La terminaison :** l'algorithme termine en un temps fini pour toute donnée en entrée.
  - **La correction partielle :** l'algorithme, lorsqu'il termine, renvoie le résultat attendu.
- **La correction totale d'un algorithme** est l'assurance de sa correction partielle ET de sa terminaison.
  - Pour chaque instance, il se termine en produisant la bonne réponse.
  - C'est-à-dire qu'il résout le problème posé!

La preuve de la correction partielle

- Bad news 1 : OK, c'est des maths… on doit prouver donc on fait une démonstration
- Bad news 2 : Il existe (une infinité) des algorithmes dont on ne peut prouver la correction partielle!
- Bad news 3 : Pour certains cette correction est prouvable mais tellement difficile à faire que l'on renonce à la faire!
- Good news 1 : Des outils ont été développés pour aider à construire ces preuves lorsqu'elles existent et qu'elles ne sont pas trop longues.
- Bad news 4 : Il faut malgré tout aimer les maths…

C’est quoi une preuve de correction partielle

- Montrez que :
  - partant d’hypothèses φ sur les données en entrée = *Précondition* de l’algorithme
  - on peut déduire en supposant que le programme P termine les propriétés attendues ψ sur les données en sortie = *post-condition* de l’algorithme.
  - Bref…on prouve $\{\varphi\}$ P $\{\psi\}$
- Formellement, on peut utiliser la ***logique de Hoare***.
  - On procède instruction par instruction de proche en proche depuis le début du programme.
  - Si on a pour une instruction T : de {Pré} T on cherche à obtenir la preuve de {Pré} T {Post} et ainsi de suite la {post} de T devient la précondition pour T’ l’instruction suivant T…
  - $\{s + t + 2 = (a + 1)^2\}$ s := s + t + 2; $\{s = (a + 1)^2\}$
- En pratique en IUT et pour la plupart des développeurs on raisonne de manière plus informelle…

La preuve de la terminaison

- La terminaison est garantie quand le programme n'utilise aucune boucle, ni récursivité.
- La terminaison reste garantie lorsque les seules boucles utilisées sont des boucles *for* « pures ».
  - Car on connaît de manière certaine le nombre (fini!) d'exécutions du corps de la boucle.
- Mais lorsque des boucles « pour » non « pures » ou « tantque » ou « répéter » ou des récursivités sont utilisées, il faut « prouver » la terminaison.

Prouver la terminaison d'une boucle

- Bad news 1 : OK, c'est encore des maths… on doit prouver donc on fait une démonstration
- Bad news 2 : Il existe des algorithmes dont on ne peut prouver la terminaison!
- Bad news 3 : Pour certains cette terminaison est tellement longue et difficile que l'on ne le fait pas!
- Good news 1 : Des outils ont été développés pour aider à construire ces preuves lorsqu'elles existent.
- Bad news 4 : Il faut malgré tout (encore) aimer les maths…

C’est quoi une preuve de terminaison

- On s’acharne sur les boucles à bornes non fixes (while, for non pures, etc.)…
- On cherche à extraire de chacune de ces boucles non fixes une suite strictement monotone sur un domaine borné.
- En termes mathématiques…
  - Ordre bien fondé : Soit (A,≤) un ensemble muni d’une relation d’ordre (partielle). Cet ordre est bien fondé s’il n’existe pas de suite infinie sur A strictement décroissante.
  - Exhiber une quantité dans un ensemble muni d’un ordre bien fondé qui diminue strictement à chaque passage dans la boucle ou à chaque appel récursif.

Petit exemple… le pgcd(a,b)

```
# a,b <=0 et a>b
m:=a;
n:=b;
reste:=0;
tantque (n != 0)
    reste:=m%n;
    m:=n;
    n:=reste;
Fin_tantque
Retourne m;
```

Exemple a=30 et b=12
m=30 et n=12 (avant la boucle)
m=12 et n=6 (première boucle)
m=6 et n=0 (deuxième boucle)
Résultat 6!

Remarque 1: (IN,≤) est bien fondé car il a un minimum qui est 0!
Remarque 2 : m%n < n (le reste est strictement inférieur au diviseur)

A chaque boucle n décroit strictement
Donc comme IN est bien fondé la borne 0 sera nécessairement atteinte donc on sortira de la boucle…

Correction totale

- **La preuve formelle de la correction totale d'un algorithme est souvent trop difficile (voire impossible) donc il faut se contenter d'une preuve « molle » (qui donne une certaine confiance mais ne prouve pas!) en usant d'autres outils :**
  - **Lecture croisée (par un co-dev)**
  - **Inspection de code (via règles + co-dev)**
  - **Analyse de code (via des outils)**
  - **test**

2. La qualité (pratique) d'un algorithme

- Le fait de trouver un algorithme qui résout un problème (dont la correction totale est assurée) est souvent satisfaisant pour un théoricien.
- Mais très insuffisant pour un développeur car :
  - Il faut implanter l'algorithme dans un langage donné.
  - Le programme devra s'exécuter sur une plate-forme à ressources contraintes
  - Il devra satisfaire les utilisateurs (ergonomie, efficacité, etc.)
  - Le programme devra être maintenu… par d'autres pour s'adapter à de nouveaux besoins, corriger des erreurs, être amélioré…

Les 50% de travaux restant

- Il faut dans l'espace (infini) des algorithmes résolvant (donc correct totalement!) en trouver un de qualité et en particulier :
  - Maintenable
  - Ergonomique
  - Performant
  - Portable
  - Réutilisable
  - Sécurisé
- Tout en respectant les exigences formulées par les clients (coûts, durée de développement, Technologies, etc.)!

Les 50% restant

- Ces propriétés externes (visibles par les utilisateurs) sont les facteurs de qualité :
  - ils décrivent la réalité perçue par les utilisateurs.
- Ils sont parfois contravariants :
  - ex Performance/maintenabilité
- Il faut donc trouver un compromis
  - Généralement ce compromis est détaillé dans une section particulière du document de spécification : la spécification non fonctionnelle

Facteurs (externe) / Critères (internes)

Ce que vous faites à l'intérieur (respecter des critères de qualité dans votre code), se voit à l'extérieur (obtenir des facteurs de qualité en usage par les utilisateurs)!

Exemples de bonnes pratiques « faciles » pour obtenir des algorithmes de qualité

- **Facteurs de Maintenabilité-Réutilisabilité** : critère de lisibilité du code source
  - Veiller à l'indentation, à donner des noms pertinents de variables, au positionnement des éléments syntaxiques…
  - Respecter la `Java code conventions` (ORACLE).
  - Documenter votre code : `Javadoc`.
  - Chaque entreprise peut imposer ses règles et en imposer puis quantifier leur respect via des **outils d'audit de code**.
- **Facteur de performance** : critère de complexité temporel et spatial
  - Evaluer l'efficacité temporelle et spatiale (ressources mémoire, disque, réseau, etc.) statiquement ou dynamiquement de vos programmes.
  - Au minimum tester les cas représentatifs et les cas les « pires ». Test de montée en charge par exemple dans le cas web avec des outils de test de performance.

3. Performance d’un algorithme

- Un algorithme est performant s’il :
  - minimise les temps de calcul
  - minimise la consommation des ressources (mémoire RAM, mémoire disque, débit réseau, etc.)
- La complexité algorithmique permet de **prédire** l'évolution en temps/ressources nécessaire pour mener un algorithme à son terme en fonction de la « quantité » de données qu’il a à traiter.

Comment mesurer l'efficacité temporelle pour UNE donnée particulière?

- Le temps de calcul (en seconde) est un très mauvais indicateur.
  - Il dépend de la machine utilisée (processeur, RAM, etc.)
  - Et du contexte de l'exécution (langage, compilateur, OS, charge réseau, débit réseau, taille et charge mémoire, etc.)
  - Une mesure individuelle est non reproductible, très contextualisée et donc non fiable.
  - Il faudrait pour la même donnée faire la médiane de très nombreuses exécutions dans des contextes variés… Exemple Timeit en python

Mesurer le nombre des instructions dans un langage donné?

- Considérer le nombre des instructions à mener dans un langage particulier pour traiter cette donnée.
- Intérêt :
  - Cette mesure est indépendante de tout contexte d'exécution
  - Mais on peut à langage constant estimer le temps en seconde d'une machine à l'autre
- Difficulté 1 : quel langage choisir : niveau langage machine, niveau byte code, niveau Java, pseudo-code, plus abstrait encore?
- Difficulté 2 : toutes les instructions ne prennent pas le même temps à s'exécuter (une affectation d'entier vs le calcul d'un cosinus en Java…)

Difficulté 1 = Aucune importance!

- En règle générale la traduction d'un programme écrit dans un langage L en un programme de L' ne modifie au plus que d'un facteur constant le nombre des opérations
  - pour les langages connus ce facteur varie en gros de 1 à 1000
  - Ainsi un programme Java réclamant $3n^3+2n^2+5$ instructions pour une donnée d particulière de taille n est en n^3.
  - Il aurait une complexité en assembleur (facteur constant k) de $3k^3\mathbf{n^3}+2k^2n^2+5$ donc en n^3 également.
  - Il n'y a donc pas de changement « d'ordre de grandeur » mais équivalence à une constante multiplicative près ($\approx k^3$ ici si $n \to \infty$).

Difficulté 2 = Aucune importance!

- Le même raisonnement s'applique si chaque type d'instruction nécessite un facteur multiplicatif constant propre par rapport à une instruction type de « référence ».
  - Supposons que l'affectation d'un entier (notée I_1) soit l'opération de référence
  - et que tous les autres types I_k d'instructions prennent un temps proportionnel à un facteur f_k près
    - $n=n_1+n_2+n_3=n_1+f_2n_1+f_3n_1=(1+f_2+f_3)n_1$
  - On ne change pas la vitesse de progression en n…

Notion de complexité pour une donnée d quelconque de taille n

- On ne s'intéresse pas à l'efficacité pour une donnée particulière.
  - Pour ce tableau [4,5,2,1,5,3] temps de tri?
  - C'est un cas particulier!
- Généralement, l'efficacité est la même pour toute une classe de données qui partagent une propriété commune : leur « taille ».
  - Pour les tableaux de taille 6, temps de tri?

La taille

- La taille *n* utilisée pour « calibrer » la complexité est fonction du problème et du choix de codage des données en entrée du programme :
  - Problème de tri d'un tableau : n est le nombre d'éléments du tableau
  - Problème du nombre premier : n est le nombre des digits utilisés pour coder l'entier d
  - Problème du voyageur de commerce : n est le nombre des sommets et arêtes du graphe G.

Complexité pire, moyenne et meilleure

- Soit c(x) le nombre d'instructions nécessaires à l'exécution de l'algorithme sur la donnée x
- Soit D_n l'ensemble des données possibles en entrée de taille n.
- Soit p(x) la distribution de probabilité ($D_n \rightarrow [0,1]$) que x soit en entrée du programme
- Les complexités dans le pire, en moyenne et le meilleur des cas sont respectivement
  - **$Cpire(n)=max(\{c(x) \mid x \in D_n\})$**
  - **$Cmoy(n)=\Sigma p(x).c(x)$ sur les $x \in D_n$**
  - **$Cmeil(n)= min(\{c(x) \mid x \in D_n\})$**

Les algorithmes de complexité « fixe »

- Certains algorithmes vérifient que pour toute classe D_n le coût est fixe donc que Cpire(n)=Cmoy(n)=Cmeil(n)
  - Recherche du min et du max dans un tableau non trié!
    - C(n)=n car il faut toujours parcourir tous les éléments…
- D'autres ont des valeurs différentes :
  - Exemple : recherche d'un élément dans un tableau
    - Cpire(n)=n (le dernier élément du tableau parcouru!)
    - Cmoy(n)=(n+1)/2 (le cas moyen c'est en moyenne au milieu!)
    - Cmeil(n)=1 (c'est le premier élément du tableau parcouru!)

Les algorithmes « fixes » et non fixes

- Les algorithmes fixes ont généralement :
  - des boucles `for`
  - et aucun `if` perturbateur de boucles (dans le bloc supérieur ou dans le corps).
- Les autres ont généralement :
  - des boucles While (leur arrêt varie)
  - et/ou des if perturbant le comportement des for et/ou des while…

Comportement asymptotique SVP

- On pourrait obtenir un décompte exact du nombre des opérations pour chacune de ces 3 complexités…
- Mais c'est fastidieux et inintéressant en pratique!
- Seul est intéressant le terme dominant, celui qui dicte asymptotiquement le comportement lorsque la taille va croissante.
  - Exemple : $f(n)=23n^3+147n^2+23n-56$.
  - Lorsque $n \to \infty$, $f(n) \approx 23n^3$.

Mieux encore, oublions les constantes!

- Ce qui importe c'est de savoir lorsque l'on augmente la taille des données d'une proportion k, de quelle proportion augmente le nombre des opérations.
  - Exemple $f(n)=10n^3$ et $g(n)=23n^3$.
  - Alors $f(kn)=10K^3n^3$ donc $F(kn)/F(K)=K^3$.
  - De même $g(kn)=23K^3n^3$ donc $g(kn)/g(K)=K^3$.
- Dans les deux cas, augmenter de K la taille en entrée augmente de K^3 le nombre des opérations à exécuter.
- Donc le facteur multiplicatif du terme de plus haut degré n'est pas une caractéristique intéressante dans ce cas…
  - Ce qui importe c'est de savoir que le nombre des opérations varie selon le cube de la taille des données en entrée.

Ordre de grandeur des complexités

- L'ordre de grandeur est généralement mesuré par 3 fonctions asymptotiques de IN $\rightarrow$IR qui permettent « d'oublier » les facteurs constants :
  - O() : borne supérieure asymptotique
    - $O(g(n)) = \{ f(n) \mid \exists c \in R^{+*} \text{ et } n_0 \in N, \forall n \geq n_0, 0 \leq f(n) \leq c.g(n) \}$
  - $\Omega()$: borne inférieure asymptotique
    - $\Omega(g(n)) = \{ f(n) \mid \exists c \in R^{+*} \text{ et } n_0 \in N, \forall n \geq n_0, 0 \leq c.g(n) \leq f(n) \}$
  - $\Theta()$: borne approchée asymptotique
    - $\Theta(g(n)) = \{ f(n) \mid \exists c_1 \in R^{+*}, \exists c_2 \in R^{+*} \text{ et } n_0 \in N, \forall n \geq n_0, 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n) \}$
  - Abusivement on note f(n)=Borne(g(n)) en place de $f(n) \in$ Borne(g(n))

4. Les règles de calcul de la complexité temporelle exacte et asymptotique

- Règle de la séquentialité
  - Complexité(I1;I2;)=Complexité(I1)+Complexité(I2)
  - Θ(I1;I2)= Θ(max(Complexité(I1), Complexité(I2)))
- Règle de la conditionnelle
  - Complexité(if (C) I1 else I2) = soit Complexité(C) +Complexité(I1) soit Complexité(C) + Complexité(I2) selon le chemin pris pour la donnée considérée
  - Θ(if (C) I1 else I2) = Θ(max(Complexité(C), Complexité(I1), Complexité(I2))) valable pour toute donnée!

Les règles de calcul du `For`

- `For (int i=0 ; i < n ; i++) {I1;}`
  - respectant
    - La boucle est « pure » : I1 n'a pas d'impact sur le compteur i et l'indice de fin n
    - La boucle est franche : ne contient pas d'instruction de continuation ou de sortie anticipée de boucle (break ou continue).
- Complexité(for) = n*(Complexité(I1) +2)+2
- Θ(n*Complexité(I1))

Les règles de calcul du `While`

- `While(C){I1;}`
- Il faut calculer dans le contexte le nombre d'itérations effectuées : n
- Complexité(While) = n*(Complexité(I1)) + (n+1)*Complexité(C)
- Θ(n*Max(Complexité(C), complexité(I1)))

5. Complexité temporelle, allons plus loin…

- La difficulté d'un problème (son coût, sa « complexité essentielle ») est intrinsèque au problème.
- On sent bien que certains problèmes sont par nature « plus difficiles » que d'autres indépendamment de toute solution d'automatisation (algorithme).
  - Tri (simple)
  - Voyageur de commerce (difficile)

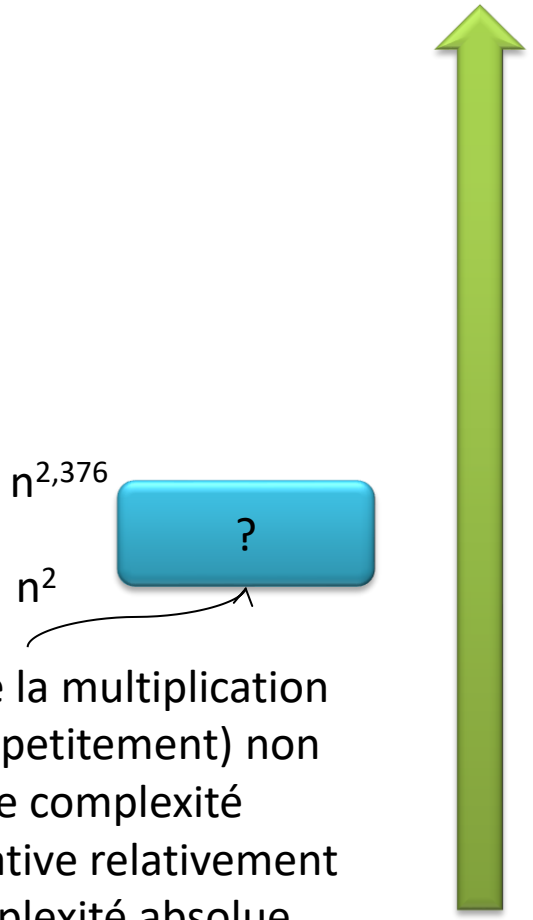
Complexité des Problème vs complexité des algorithmes

- Un problème calculable (spécifié par une fonction f) est associé à une infinité d'algorithmes.
  - La complexité va permettre de comparer tous ces algorithmes dans un modèle de calcul donné selon les ressources qu'ils consomment.
- La complexité relative d'un problème est celle de l'algorithme le plus efficace connu à ce jour pour le résoudre dans le pire des cas.
  - Exemple : Multiplication de deux matrices n×n : algorithme de Coppersmith-Winograd en $O(n^{2,376})$ en temps dans le modèle de Turing.
  - C'est un **la borne supérieure (potentiellement temporaire) de ce qui est connu.**
- la complexité absolue d'un problème est fournie par une étude théorique indépendante de tout algorithme se basant sur l'essence même du problème dans le pire des cas.
  - Exemple : le tri est prouvé comme étant en $\Theta(n\log_2(n))$ en temps dans le pire des cas dans le modèle de Turing.
  - C'est une **borne à la fois inférieure et supérieure définitive** (dans un modèle de calcul)!
  - Il suffit de trouver ensuite un algorithme ayant ce niveau de complexité pour dire que le problème est **résolu** (à une constante multiplicative près!)
- La complexité plancher d'un problème est fournie par une étude théorique qui montre que toute solution a, au moins, ce niveau de complexité dans le pire des cas
  - C'est la **borne inférieure** (**potentiellement temporaire)** dans un modèle de calcul donné avec les connaissances actuelles.
  - Cela signifie qu'il n'y a aucun espoir de découvrir un algorithme de complexité strictement inférieure dans l'avenir.

Problèmes non résolus

- Dans un modèle de calcul donné, il y a donc des problèmes :
  - Résolus :
    - avec une complexité absolue prouvée et un algorithme connu de ce niveau de de complexité
  - Non résolus :
    - Sans complexité absolue établie et donc avec une complexité relative et plancher plus ou moins « proche »
    - Plus l'écart est grand, plus l'indétermination est grande!

Catalogue des problèmes…



$2^n n^2$

?

Le problème du voyageur de commerces est (hautement!) non résolu. Les complexités relatives et planchers sont très éloignées et incertaines…

$n^{2,376}$

?

n^2

Le problème de la multiplication matricielle est (petitement) non résolu. Il y a une complexité plancher et relative relativement proche. La complexité absolue est, elle, inconnue…

n

Le problème du tri est prouvé de complexité absolue en $n.\log_2 n$. On a trouvé plusieurs algorithmes de ce niveau de complexité : le problème est donc résolu

Les classes de complexité usuelles

	Notation	Type de complexité
Facile	$O(1)$	complexité constante (indépendante de la taille de la donnée)
	$O(\log(n))$	complexité logarithmique
	$O(\sqrt{n})$	complexité racinaire
	$O(n)$	complexité linéaire
	$O(n\log(n))$	complexité linéarithmique
Raisonnable	$O(n^2)$	complexité quadratique
	$O(n^3)$	complexité cubique
	$O(n^p)$	complexité polynomiale
Déraisonnable	$O(e^n)$	complexité exponentielle
	$O(n!)$	complexité factorielle
	$O(2^{2^n})$	complexité doublement exponentielle

Exemples de classes de complexité

Notation	Type de complexité	Temps pour n = 5	Temps pour n = 10	Temps pour n = 20	Temps pour n = 50	Temps pour n = 250	Temps pour n = 1 000	Temps pour n = 10 000	Temps pour n = 1 000 000	Problème exemple
$O(1)$	complexité constante	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	Accès tableaux
$O(\log(n))$	complexité logarithmique	10 ns	10 ns	10 ns	20 ns	30 ns	30 ns	40 ns	60 ns	Dichotomie
$O(n)$	complexité linéaire	50 ns	100 ns	200 ns	500 ns	2.5 µs	10 µs	100 µs	10 ms	Parcours de liste
$O(n\ log^*(n))$	complexité quasi-linéaire	50 ns	100 ns	200 ns	501 ns	2.5 µs	10 µs	100,5 µs	10 050 µs	Triangulation de Delaunay
$O(n\log(n))$	complexité linéarithmique	40 ns	100 ns	260 ns	850 ns	6 µs	30 µs	400 µs	60 ms	Tris dont le Tri fusion ou le Tri par tas
$O(n^2)$	complexité quadratique (polynomiale)	250 ns	1 µs	4 µs	25 µs	625 µs	10 ms	1 s	2.8 heures	Parcours de tableaux 2D
$O(n^3)$	complexité cubique (polynomiale)	1.25 µs	10 µs	80 ms	1.25 ms	156 ms	10 s	2.7 heures	316 ans	Multiplication matricielle non-optimisée.
$O(n^{\log(n)})$	complexité sous-exponentielle	30 ns	100 ns	492 ns	7 µs	5 ms	10 s	3.2 ans	10^{20} ans	
$O(e^n)$	complexité exponentielle	320 ns	10 µs	10 ms	130 jours	10^{59} ans	...	...	...	Problème du sac à dos par force brute.
$O(n!)$	complexité factorielle	1.2 µs	36 ms	770 ans	10^{48} ans	...	...	...	...	Problème du voyageur de commerce (avec une approche naïve).
$O(2^{2^n})$	complexité doublement exponentielle	4.3 s	10^{278} ans	...	...	...	...	...	...	Décision de l'arithmétique de Presburger

Source wikipedia – pour un temps d'exécution des opérations élémentaires unique de 10ns

Impact de la taille des données

- Il y a des sauts réels, en « ordre de grandeur » d'écart entre ces classes de complexité.
  - Le passage de l'une à l'autre n'est pas simplement une multiplication par 10 ou par 100, mais par beaucoup plus. On change radicalement la taille des données traitables!
- Affreux ! Il existe une classe de complexité « extrême » dans laquelle se trouve des problème calculables mais dont la complexité temporelles est pire que toute complexité n-exponentielle (on empile n exponentiation)!
- Attention en complexité on est toujours à un facteur multiplicatif prêt :
  - certains algorithmes en $\Theta(n^2)$ peuvent donc, sur de petites données, travailler plus vite que d'autres en $\Theta(n\log n)$
  - Certains algorithmes peuvent dans le cas moyen ou dans certaines situations se révéler plus rapide que d'autres algorithmes pourtant de complexité plus faible dans le pire des cas…

Calculabilité théorique vs pratique

- Bad news : De nombreux problèmes intéressants sont donc calculables en théorie mais non en pratique au-delà de certaines tailles de donnée.
  - soit on ne fait pas le calcul au-delà d'une certaine taille
  - soit on se contente d'un calcul plus rapide mais approché (par un minimum local dont on sait parfois borner la marge d'erreur).
- Good news! On utilise cette incapacité de calcul en pratique pour définir des stratégies de cryptographie rapides (en temps polynomial si on a la clé) mais inviolables (car en temps exponentiel si on ne l'a pas).

Cartographie des problèmes dans un modèle Turing-complet