Complexités

Première question théorique :
complexités en temps et en espace.

Motivations

Contexte

- un problème à résoudre
  - un problème = une question + des paramètres (données "en entrée")
- une instance de ce problème qui admet au moins une solution
  - une instance = un choix des données d'entrée
- un algorithme qui calcule sa solution
  - un ou même plusieurs algorithmes

Questions du jour

- De combien de temps a besoin l'algorithme pour calculer cette solution ?
- De combien d'espace-mémoire a besoin l'algorithme pour calculer cette solution ?

Motivations

- Problème se résout-il en
  - un temps raisonnable ?
  - avec des ressources raisonnables ?
- Comment quantifier *l'efficacité* de l'algorithme ?
  - Objectif : choisir, adapter, améliorer, …

Je calcule la somme de n valeurs entières

Une instance : un choix de n et des n valeurs du tableau t

Principe d'un algorithme : je parcours le tableau, "du début à la fin", je lis chaque valeur, je l'accumule dans une variable (initialement mise à zéro), je retourne cette variable.

```
def sommer(t, dim_t):
    ''' somme itérative de n=dim_t valeurs entières stockées dans un tableau t
    entrées. t tab d'int de longueur dim_t
    retourne. res int'''
    res = 0     # j''accumule dans res
    for i in range(dim_t):
        res = res + t[i]
    return res
```

Complexité en temps, complexité en espace

Quel est le temps nécessaire à la résolution du problème avec un algorithme ?

Quel est l'espace-mémoire nécessaire à la résolution du problème avec un algorithme ?

Complexité ? en temps ? en espace-mémoire ?

1. On a un problème à résoudre :
   - calculer la somme de n valeurs entières stockées dans un tableau t

2. On a (au moins) un algorithme qui résout ce problème
   - par exemple: `sommer` dans sa version "boucle for"

3. Combien de temps pour calculer la réponse ?
   - ça dépend de n : le nombre de valeurs à sommer
   - A priori, le temps de calcul est une fonction croissante de n
   - OK mais croissante comment ? linéaire ? quadratique ? logarithmique ?
   - on parle aussi de **coût** de l'algorithme

4. Combien d'espace-mémoire pour calculer la réponse ?
   - ça dépend aussi de n : faut déjà réussir à stocker les n valeurs !
   - OK mais quel est l'espace mémoire supplémentaire pour résoudre le problème ?
   - Dépendant ou indépendant de n ? dépendant comment ?

Paramètres de la complexité

On a un problème à résoudre :

- calculer la somme de n valeurs entières stockées dans un tableau t

Quels sont les paramètres du coût de sa résolution ?

1. La taille du problème
   - ce qui caractérise la taille dépend du problème
   - sur l'exemple de la somme :
     taille = n : le nombre de valeurs entières à sommer
   - je trie n valeurs entières :
     taille = n : le nombre de valeurs entières à trier
   - je calcule l'addition de 2 nombres entiers :
     - la taille est le nombre de chiffres de chaque opérande (ou le max des 2)

2. Certaines données/entrées du problème : instance du pb
   - je trie une liste déjà triée *vs*. je trie une liste "très bien mélangée"

complexité = f(taille)
ordre de grandeur pour des très grandes tailles

complexité = f(taille)

- un seul paramètre : la taille du problème
- la complexité (le coût) sera une fonction de ce paramètre
- le paramètre est donc ce qui fait varier le temps/l'espace mémoire nécessaire à la résolution du problème

Complexité asymptotique

- on s'intéresse souvent au coût pour des tailles arbitrairement grandes
- l'ordre de grandeur de f est caractéristique de ce coût pour des grandes tailles : linéaire, quadratique, logarithmique, semi-log, exponentiel

Pire cas, meilleur cas, cas moyen

- si la difficulté / le coût du calcul de la solution dépend de l'instance du problème, on peut distinguer :
  - le pire des coûts
  - le meilleur des coûts
  - le coût moyen (c'est plus difficile : la moyenne sur quel échantillon ?)

chacun de ces coûts étant toujours une fonction de la (grande) taille du problème.

Comment mesurer le coût d'un algorithme ?

On a (au moins) un algorithme qui résout ce problème

- `sommer` dans sa version itérative "boucle for"

On veut mesurer les coûts en temps d'exécution et en espace-mémoire. Comment exécuter un algorithme ?

Analyse de complexité

On fait exécuter l'algorithme par **un modèle de machine**

- simple : on cache beaucoup de détail d'une véritable exécution
- mais pas trop simpliste : les résultats et les observations réelles sont raisonnablement corrélées
- l'exécution continue a dépendre des données d'entrées ; pire cas, meilleur cas
- un premier objectif : dégager des tendances, des ordres de grandeur

Complexité en temps

Analyse du pire temps d'exécution avec le modèle RAM ?

Et en pratique ?

Un modèle de complexité pour mesurer le coût en temps d'un algorithme

Toutes les instructions importantes comptent 1 unité de temps :

1 = affectation = comparaison = opération arithmétique = op. logique = accès mémoire = entrées/sorties

Séquentialité des instructions :

si l'instruction p coûte c1 et l'instruction q coûte c2,
alors la suite d'instructions p, q coûte c1+c2

Coût du if... elif ... else ...:

inférieur ou égal au coût maximum de chaque branche d'instructions

Coût de la boucle for i in range(n): p

si le coût de p ne dépend pas de i : n x coût(p)
sinon : la somme des coûts de chacune des n répétitions

Coût de la boucle while(condition): p

dépend du nombre de répétitions, inconnu a priori
on peut cependant majorer ce nombre de répétitions

Je calcule la somme de n valeurs entières

Somme itérative avec accumulation

- une boucle for
- pas de test if …
- additions entières, affectations d'entiers, accès (lecture) éléments d'un tableau, retour, contrôle de boucle for

```
1   def sommer(t, n):
        ''' somme itérative de n valeurs entières stockées dans un
        tableau t. entrées. t tab d'int de longueur n
        retourne. res int'''
2       res = 0          # j''accumule dans res
3       for i in range(n):
4               res = res + t[i]
5       return res
```

Comment mesurer le coût d'un algorithme ?

On a (au moins) un algorithme qui résout ce problème

- `sommer` dans sa version itérative "boucle for"

On a un modèle de complexité basé sur :

- chaque instruction compte 1 unité (de temps)
  - 1 = affectation = comparaison = opération arithmétique = op. logique = ...
  - donc on pourrait/devrait tout compter ...

En pratique, on choisit certaines instructions *significatives* du temps de traitement de l'algorithme

- on compte "seulement" les additions dans `sommer`
- on a autant d'affectations dans `res` que d'additions
- on ne compte pas les affectations et les additions cachées dans la mise à jour des indices de boucles pour
- conclusion : complexité(`sommer`) = f(nombre d'additions du corps de boucle)

C(sommer) = f(nombre d'additions des lignes L3-4)

```
3.  for in in range(n):
4.      res = res + t[i]
```

Comptons !

- n est la taille du problème "sommer n valeurs"
- l'algorithme sommer effectue 1 addition (L4) à chacune des n répétitions de la boucle pour (L3-4)
- **complexité de la sommation séquentielle sommer : C(n) = n**
- L'algorithme sommer a une **complexité linéaire** en la taille du problème à résoudre

Interprétation :

- si on double le nombre de valeurs à sommer, on double le temps de calcul

- C'est d'autant plus vrai que n est assez grand pour que le temps de ces opérations (les additions) constitue *la part significative* du temps total de l'exécution de sommer.

Du modèle pour l'algorithme à la mesure d'un « vrai » programme

Des différences importantes entre le modèle d'analyse de complexité de l'algorithme et la chaîne actuelle de calcul : processeur, mémoires hiérarchiques, options du compilateur, parallélisme, prédiction ...

La mesure des performances d'un code est un processus expérimental assez difficile et qui ment facilement

L'analyse de complexité est cependant significative de la tendance des mesures

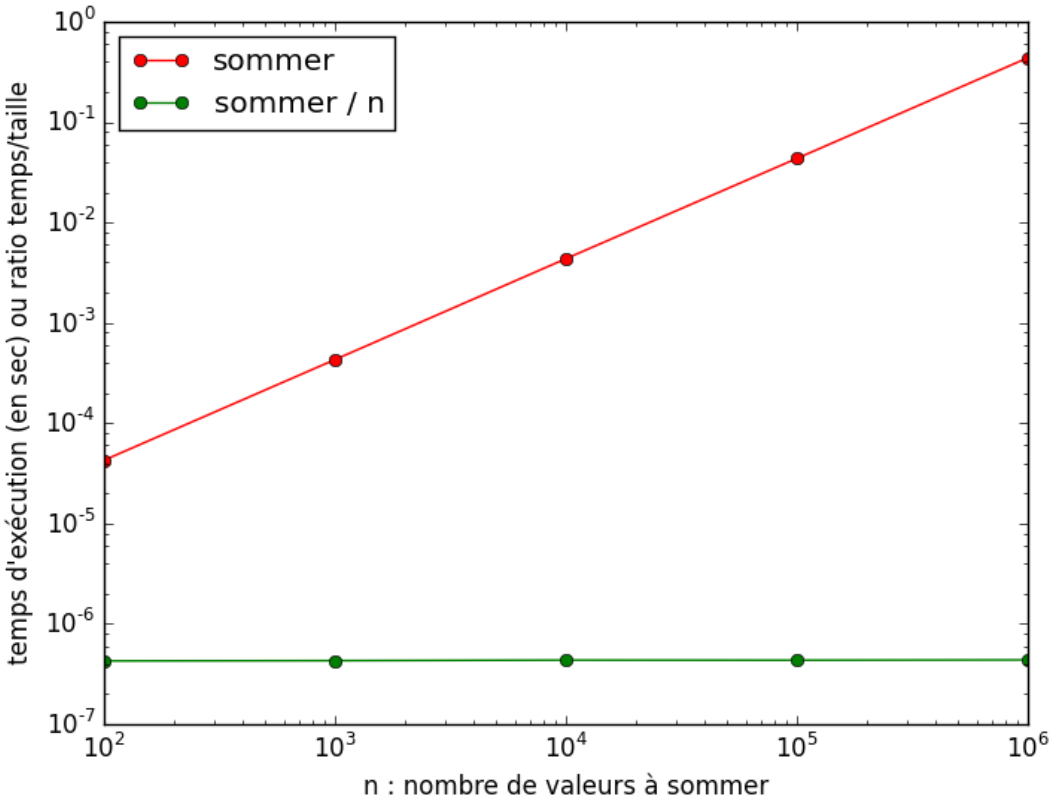
en pratique : un problème donné de grande taille est résolu plus rapidement par un algorithme en n, que par un algorithme en n^2 , et encore plus que par un algorithme en n^3 ...

Exemple pour des algorithmes très calculatoires

- on compte le nombre d'opérations arithmétiques
- on mesure les temps d'exécution d'un programme (python sur mac-intel)

Des mesures réelles de sommer

Mesures sur ma machine de la somme codée en C :

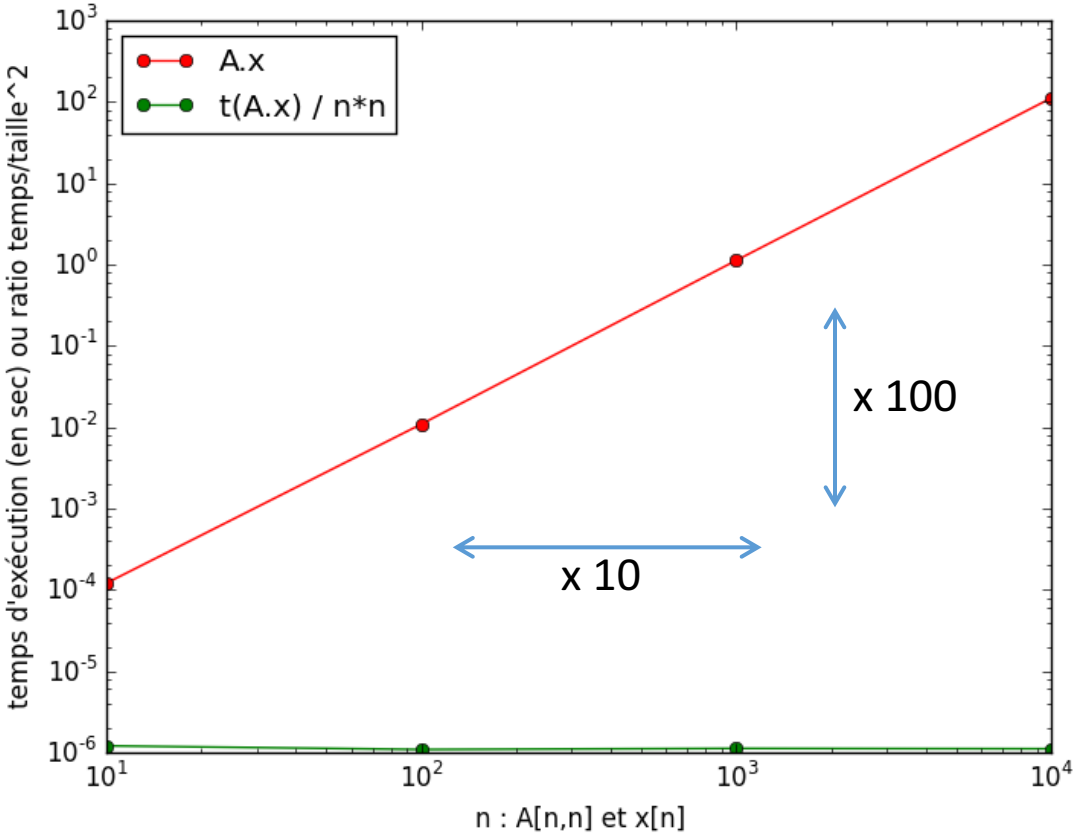


sommer est bien **linéaire** en nombre d'additions

Quel sur-coût observe-t-on quand la taille du problème est multipliée par 10 ?

Des mesures réelles de $\mathtt{A\,x}$

Le produit matrice-vecteur $\mathtt{A\,x}$ est **quadratique** en nombre d'opérations arithmétiques



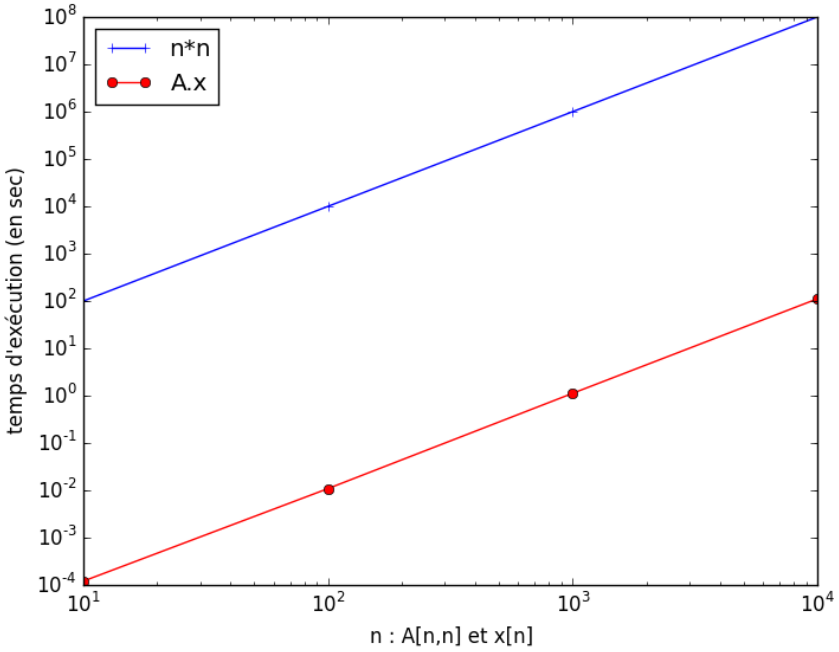
Mesures du temps t(A,x) de l'exécution du produit matrice-vecteur codé en python sur ma machine

Attention : échelles logarithmiques sur les 2 axes !

- se convaincre que la pente de t(A.x) vaut 2, ce qui représente $k \cdot n^2$
- la droite $t(A.x)/n^2$ est horizontale : ce ratio est constant

Des mesures réelles de `A x`

Le produit matrice-vecteur `A x` est quadratique en nombre d'opérations arithmétiques



Attention : échelle $\log_{10}$ sur les axes des x et des y !

La pente vaut 2, ce qui représente 10^2

Autre illustration :
le temps de calcul représenté en échelle log-log est une droite parallèle à celle de n^2

Conseil : être à l'aise pour choisir le tracé le plus parlant !

Complexité un peu plus avancée

Complexité et log

Réduire la complexité

Multiplier 2 entiers, épisode 1

Pas que des puissances entières de la taille

Problème : rechercher si une valeur est présente au non dans un tableau de n valeurs triées

Mesure de complexité : **nombre de comparaisons**

Un algorithme de **complexité linéaire** : recherche séquentielle

- je parcours le tableau du début à la fin et je compare chaque valeur du tableau à la valeur cherchée
- j'effectue entre 1 et n comparaisons, au plus n, jamais plus que n
- $C_{\text{recherche séquentielle}}(n) = n$ (ou k x n ou plus tard O(n))
- **Nombre maximal de comparaisons = nombre d'itérations du pire cas**

Un algorithme de **complexité logarithmique** : la recherche dichotomique

- je partage le tableau en 2 et je compare la valeur du milieu,
- selon le résultat de la comparaison je jette la moitié droite (ou gauche) du tableau,
- je **recommence** sur ce tableau de taille moitié : partage en 2, comparaison milieu, abandon d'une moitié
- **et ainsi de suite** jusqu'à

  avoir trouvé la valeur et là, je m'arrête

  OU obtenir un (sous-)tableau réduit à 0 ou 1 élément et là, je m'arrête ... peut-être sans l'avoir trouvé

Nombre de comparaisons = nombre de découpages en 2

Nombre **maximal** de comparaisons = nombre de découpages en 2 jusqu'à l'arrêt "avec absence" (pire cas)

Combien de découpages par 2 de n valeurs jusqu'à en obtenir 1 seule ?

La réponse : $\log_2(n)$

Le principe : partons de $n = 2^p$

- division 1 → $2^p / 2 = 2^{p-1}$
- division 2 → $2^{p-1} / 2 = 2^{p-2}$
- …
- division k → $2^{p-k+1} / 2 = 2^{p-k}$ (la propriété à prouver par récurrence)
- …
- division p → $2^{p-p+1} / 2 = 2^{p-p} = 2^0 = 1$

Il faut p divisions par 2 pour passer de 2^p à 1

et $p = \log_2(2^p) = \log_2(n)$

$C_{\text{rech. dichotomique}}(n) = \log_2(n)$ ou : $k \times \log_2(n) = k' \times \log_{10}(n) = k'' \times \ln(n)$, ou plus tard $O(\log(n))$

La recherche dichotomique : exemple de complexité logarithmique

Les algorithmes issus de stratégie "diviser pour régner" introduisent des complexités en log : log(n) ou n.log(n) ou ...

Réduire la complexité ?

Diviser pour régner ou *divide and conquer*

- principe général basé sur la **récursivité**
- réduire le problème en un problème similaire ET de taille réduite ...

    ... recommencer cette réduction ....

        ... jusqu'à obtenir un problème suffisamment petit pour pouvoir trouver sa solution "immédiatement",

    à partir de cette solution, construire la solution du problème plus grand ...

... et ainsi de suite jusqu'à obtenir la solution du problème de départ

- principe présent dans la recherche dichotomique !

Un petit zoom sur le produit de 2 entiers

Problème : multiplier 2 entiers de n chiffres (en base 10)

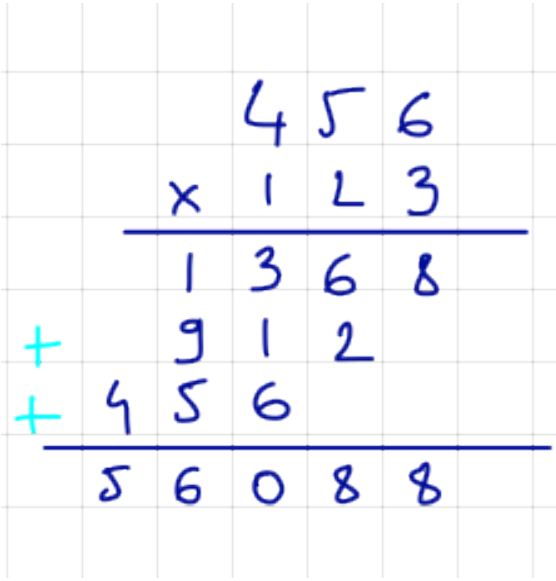
Mesure : nombre de multiplications "chiffre à chiffre"

Algorithme (dit) naïf de multiplication :

A la main ... pour sentir la réponse

chaque chiffre d'un opérande (y'en a n) est multiplié par chaque chiffre de l'autre opérande (y'en a aussi n) donc au total : n x n = n^2 multiplications "chiffre à chiffre"

La multiplication naïve est quadratique



$3 \times 456 = 3 \times (6 + 5 \times 10 + 4 \times 100)$
$= 3 \times 6 + (3 \times 5) \times 10 + (3 \times 4) \times 100$
$= 18 + 15 \times 10 + 12 \times 100$
$= 18 + 150 + 1200$
$= 1368$ ✓

$123 \times 456 = 3 \times 456 + 2 \times 456 \times 10 + 1 \times 456 \times 100$

Décompte des opérations :

- une multiplication 1 chiffre x 3 chiffres = 3 multiplications (+ 2 additions)
- on en fait 3, soit 9 multiplications (+ 6 additions)
- et on ajoute les 3, soit 9 multiplications (et 9 additions)

et ce pour la multiplication de 2 nombres à 3 chiffres :

on a bien effectué $3^2 = 9$ multiplications

Remarque : on ne compte pas les multiplications par les puissances de la base : 1, 10, 100, ...
ni les additions des produits partiels

Complexité quadratique du produit de 2 entiers

Analyse : formalisons le produit a x b en base 10 ICI : **n+1** chiffres

$$
\begin{aligned}
a &= (a_n a_{n-1} \dots a_2 a_1 a_0)_{10} \\
&= a_0 + a_1 \cdot 10 + a_2 \cdot 10^2 + \dots + a_n \cdot 10^n \\
b &= (b_n b_{n-1} \dots b_2 b_1 b_0)_{10} = \sum_{i=0}^{n} b_i \cdot 10^i \\
a \times b &= (a_n \dots a_1 a_0) \times (b_n \dots b_1 b_0) \\
&= a_0 \times b_0 + (a_1 \times b_0 + a_0 \times b_1) \cdot 10 \\
&\quad + (a_2 \times b_0 + a_1 \times b_1 + a_0 \times b_2) \cdot 10^2 + \dots \\
&\quad + (a_k \times b_0 + a_{k-1} \times b_1 + \dots + a_0 \times b_k) \cdot 10^k + \dots \\
&\quad \dots + (a_n \times b_{n-1} + a_{n-1} \times b_n) \cdot 10^{2n-1} + (a_n \times b_n) \cdot 10^{2n}
\end{aligned}
$$

Comptons les produits partiels (les x) :

1 + 2 + 3 + … + k + … (n-1) + n + n+1 +n + (n-1) + (n-2) + … + 2 + 1

= (n+1)(n+2)/2 + n(n+1)/2 = $(n+1)^2$ rappel : ici a et b ont n+1 chiffres

Réduire la complexité ?

Problème : multiplier 2 entiers de n chiffres (en base 10)

Mesure : nombre de multiplications "chiffre à chiffre"

Algorithme naïf de multiplication : complexité quadratique

n^2 multiplications "chiffre à chiffre"

Existe-t-il un algorithme qui calcule le produite de 2 entiers et qui soit d'une complexité inférieure à n^2 ?

Algorithme de Karatsuba (1960) : complexité en $n^{log_2 3} \approx n^{1.58}$

si n = 1000 :

- multiplication naïve = 1 000 000 produits
- multiplication de Karatsuba = 50 000 produits, soit 20 fois plus rapide !!!

→ Sera présenté en exercice sur la récursivité !

Un petit zoom sur le produit de 2 entiers

Remarques pour finir

- Démarche similaire en base 2 : $$\begin{aligned} a &= (a_n a_{n-1} \dots a_2 a_1 a_0)_2 \\ &= a_0 + a_1 \cdot 2 + a_2 \cdot 2^2 + \dots + a_n \cdot 2^n \end{aligned}$$

- Algorithme naïf de multiplication :

  produit par la base = décalage d'un chiffre/bit vers la gauche

  produit axb : suite d'additions (de prod. ch. à ch.) et de décalages

  similaire au produit de polynômes

- Algorithmes rapide (Karatsuba)

  entier sur machine = 64 bits et produit effectué "en" matériel

  intérêt de Karatsuba : multiplier des grands entiers : N = n0 + n1 + ...

  GMP :

  https://gmplib.org/

  «Arithmetic without limitations»

  The GNU Multiple Precision Arithmetic Library

  sage (python)

  http://www.sagemath.org/fr/

Premières conclusions sur la complexité en temps

Un modèle de complexité

- c'est juste un modèle : très approximatif mais significatif mais de ce qui se passe "en vrai" ...
- mais en informatique : ce qu'on rencontre en vrai aujourd'hui aura peut-être disparu demain (calcul quantique)

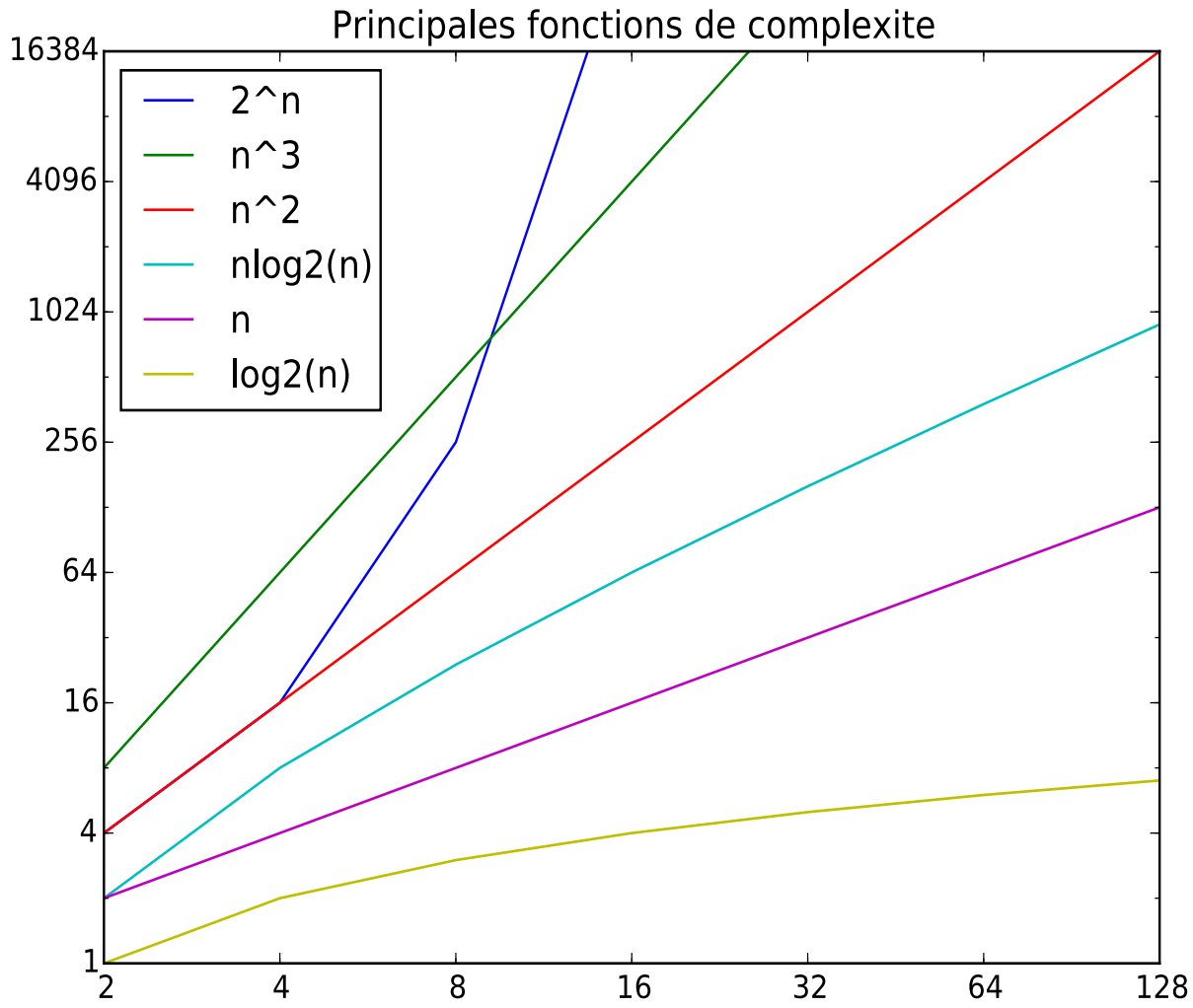
Ce qui importe : c'est l'ordre de grandeur du coût mesuré comme une fonction de la taille du problème, quelque soit l'instance du pb.

Ce qui parle : c'est le surcoût de temps pour résoudre un problème deux fois plus gros, dix fois plus gros !

10 fois plus gros avec une complexité :

- cubique = 1000 fois plus long
- quadratique = 100 fois plus long
- linéaire = 10 fois plus long ... à la rigueur
- racine carrée = environ 3 fois plus long ... oui !
- logarithmique = 2 fois plus long ... oui : je veux !
- exponentiel = 10^{10} fois plus long ... aie aie aie : trop cher pour moi !!!!

Pour conclure sur la complexité en temps



Complexité en temps, complexité en espace

Quel est le temps nécessaire à la résolution du problème avec un algorithme ?

Quel est l'espace-mémoire nécessaire à la résolution du problème avec un algorithme ?

Et sur la complexité en espace-mémoire

Quelle quantité d'espace-mémoire est nécessaire pour que l'algorithme trouve la solution du problème ?

Quel espace-mémoire mesurer ?

- on ne compte pas la place des données d'entrée, ni des résultats : incompressible quelque soit l'algorithme
- on compte "juste" la place supplémentaire

Cas facile / moyen / un peu difficile :

- facile = statique : toutes les variables utilisées sont connues "dans l'algo"
  on compte leurs places selon leurs types : scalaire, tableau 1D, 2D ...
- moyen = dynamique
  on utilise de l'allocation dynamique de mémoire (cf. cours de C en L2)
- un peu difficile = appels récursifs
  l'algorithme est récursif ... à venir très bientôt !!
  la complexité en espace-mémoire peut alors être très, voire trop importante

Complexité en espace-mémoire de sommer

```
def sommer(t, n):
    ''' somme itérative de n valeurs entières stockées dans un tableau t
    entrées. t tab d'int de longueur n
    retourne. res int'''
    res = 0
    for i in range(n):
        res = res + t[i]
    return res
```

Analyse

. on ne compte pas les n places-mémoire pour l'entrée : le tableau d'entiers t

. on ne compte pas res qui est le résultat retourné

. il suffit de pouvoir stocker l'accumulation des t[i] : ici dans res

. remarque : c'est toujours la même ligne qui compte !

. un seul entier suffit et ce quelque soit la taille du problème !

Conclusion :

. la complexité en espace-mémoire de sommer est constante (et égale à 1).

Synthèse de la séance

Analyse de complexité

Une sensibilisation à une première notion (un peu) théorique

Objectifs : choisir l'algorithme le plus efficace, estimer la taille d'un problème qu'on peut résoudre en un temps raisonnable

A venir ce semestre : terminaison, correction, preuves

Complexités en temps et en espace

- c'est le début : comprendre le principe et l'objectif général
- comprendre le *modèle* **d'analyse de complexité**
- identifier ce qu'il est significatif de mesurer, le sens de la mesure
- quel est le sur-coût quand je double la taille de mon problème ?

Pertinence pratique de l'étude menée ?

modèle simple vs. exécution machine complexe

lecture / écriture mémoire : tout sauf du temps constant

opération arithmétique : addition vs. division, entier vs. flottant

exécution séquentielle des instructions : ça n'existe plus (ou presque plus)

les machines actuelles exécutent plusieurs instructions en parallèle, dans un ordre différent du programme, elles spéculent sur le résultats de tests ou d'accès mémoire ...

les estimations asymptotiques (grande taille du pb) masquent ces effe

Analyse de complexité

Etudier la complexité, c'est ... complexe

Théorie de la complexité : une branche de l'informatique

Objectif : classer les problèmes selon leurs difficultés de résolution, (cad. le coût de leur résolution quelque soit l'algorithme utilisé) et les relations entre ces classes de problèmes

Modèles d'exécution : RAM (ici, Random Access Memory), machine de Turing ("dite" déterministes ou non déterministes), automates, ...

Objectif : réduire la complexité

- pour pouvoir résoudre des gros problèmes
- ou des problèmes pas nécessairement gros mais compliqués

Il existe de nombreux problèmes qu'on se sait pas résoudre exactement en un temps réaliste

complexité exponentielle = 2^n → problème du sac à dos (brute force)

complexité factorielle : n! → problème du voyageur de commerce (naïf)

(source wikipedia, temps de base = 10 nanosec)

exponentielle : pour n = 50, temps T = 130 jours, pour n= 250 = 10^{59} ans

factorielle : pour n = 50, temps T = 10^{48} ans

A venir : vers une vision plus formelle

Ce qui importe : c'est l'ordre de grandeur du coût mesuré comme une fonction de la taille du problème, quelque soit l'instance du pb.

Ce qui parle : c'est le surcoût de temps pour résoudre un problème deux fois plus gros, dix fois plus gros !

10 fois plus gros avec une complexité :

- *cubique = 1000 fois plus long*
- *quadratique = 100 fois plus long*
- *linéaire = 10 fois plus long ... à la rigueur*
- *racine carrée = environ 3 fois plus long ... oui !*
- *logarithmique = 2 fois plus long ... oui : je veux !*
- *exponentiel = 10^{10} fois plus long ... aie aie aie : trop cher pour moi !!!!*

Formaliser l'analyse de la complexité asymptotique

Formaliser pour pour dégager plus facilement les tendances asymptotiques

- au pire **aussi rapide** qu'un algorithme quadratique ?
- au pire **au moins** aussi rapide qu'un algorithme quadratique ?
- au pire **au plus** aussi rapide qu'un algorithme quadratique ?

Comparaison asymptotique de fonctions et notations de Landau

- équivalent asymptotique : $\theta(n), \theta(n^2), \theta(n\log(n))$
- minorant asymptotique : $\Omega(n), \ldots$
- **majorant asymptotique** : $\mathcal{O}(n), \ldots$