Partie 2

Outils d'analyse

Plan

1. Correction d'algorithmes
2. Complexité algorithmique
3. Résolution de sommations et de récurrences

Plan

Analyse d'algorithmes

Questions à se poser lors de la définition d'un algorithme :

- Mon algorithme est-il correct ?
- Mon algorithme est-il efficace ?

Autres questions importantes seulement marginalement abordées dans ce cours :

- Modularité, fonctionnalité, robustesse, facilité d'utilisation, temps de programmation, simplicité, extensibilité, fiabilité, existence d'une solution algorithmique...

Correction d'un algorithme

- La correction d'un algorithme s'étudie par rapport à un problème donné
- Un problème est une collection d'instances de ce problème.
  - Exemple de problème : trier un tableau
  - Exemple d'instance de ce problème : trier le tableau $[8, 4, 15, 3]$
- Un algorithme est correct pour une instance d'un problème s'il produit une solution correcte pour cette instance
- Un algorithme est correct pour un problème s'il est correct pour toutes ses instances (on dira qu'il est totalement correct)
- On s'intéressera ici à la correction d'un algorithme pour un problème (et pas pour seulement certaines de ses instances)

Comment vérifier la correction ?

- Première solution : en testant concrètement l'algorithme :
  - Suppose d'implémenter l'algorithme dans un langage (programme) et de le faire tourner
  - Suppose qu'on peut déterminer les instances du problème à vérifier
  - Il est très difficile de prouver empiriquement qu'on n'a pas de bug
- Deuxième solution : en dérivant une preuve mathématique formelle :
  - Pas besoin d'implémenter et de tester toutes les instances du problème
  - Sujet à des "bugs" également
- En pratique, on combinera les deux

- Outils pour prouver la correction d'un algorithme :
  - Algorithmes itératifs : triplets de Hoare, invariants de boucle
  - Algorithmes récursifs : preuves par induction

Assertion

- Relation entre les variables qui est vraie à un moment donné dans l'exécution
- Assertions particulières :
  - Pre-condition P : conditions que doivent remplir les entrées valides de l'algorithme
  - Post-condition Q : conditions qui expriment que le résultat de l'algorithme est correct
- P et Q définissent resp. les instances et solutions valides du problème
- Un code est correct si le triplet (de Hoare) $\{P\}$ code $\{Q\}$ est vrai.
- Exemple :

$$\{x \geq 0\} y = \mathrm{SQRT}(x) \{y^2 == x\}$$

Correction : séquence d'instructions

$\{P\}$
S1
S2
...
Sn
$\{Q\}$

Pour vérifier que le triplet est correct :

- on insère des assertions P_1, P_2, ..., P_n décrivant l'état des variables à chaque étape du programme
- on vérifie que les triplets $\{P\}$ S1 $\{P_1\}$, $\{P_1\}$ S2 $\{P_2\}$, ..., $\{P_{n-1}\}$ Sn $\{Q\}$ sont corrects

Trois types d'instructions : affectation, condition, boucle

Correction : affectations

Le triplet suivant est correct :

$$\{Q[x \to e]\}$$
$$x = e$$
$$\{Q\}$$

$Q[x \to e]$ est obtenu en remplaçant les occurrences de x par e dans Q.

Pour prouver un triplet :

$$\{P\}$$
$$x = e$$
$$\{Q\}$$

il faut montrer que P implique $Q[x \to e]$.

Exemples : les triplets suivants sont corrects

$$\{x == 2\}$$
$$y = x + 1$$
$$\{y == 3\}$$

$$\{x == 42\}$$
$$y = x + 1$$
$$z = y$$
$$\{z == 43\}$$

Correction : conditions

$\{P\}$
if B
    C1
else
    C2
$\{Q\}$

Pour prouver que le triplet est correct, on doit prouver que

- $\{P$ et $B\}$ C1 $\{Q\}$
- $\{P$ et non $B\}$ C2 $\{Q\}$

sont corrects

Exemple :

$\{x < 6\}$
if $x < 0$
    $y = 0$
else
    $y = x$
$\{0 \leq y < 6\}$

Correction : boucles

$\{P\}$
INIT
while B
    CORPS
FIN
$\{Q\}$

$\{P\}$
INIT
$\{I\}$
while B
    $\{I$ et $B\}$ CORPS $\{I\}$
$\{I$ et non $B\}$
FIN
$\{Q\}$

Pour prouver que le triplet est correct :

- On met en évidence une assertion particulière I, appelée invariant de boucle, qui décrit l'état du programme pendant la boucle.
- On prouve que :
  - $\{P\}$ INIT $\{I\}$ est correct
  - $\{I$ et $B\}$ CORPS $\{I\}$ est correct
  - $\{I$ et non $B\}$ FIN $\{Q\}$ est correct

Si on a plusieurs boucles imbriquées, on les traite séparément, en démarrant avec la boucle la plus interne.

Correction : terminaison de boucle

```
INIT
while B
      CORPS
FIN
```

- Un fois qu'on a prouvé que le triplet était correct, il faut encore montrer que la boucle se termine
- Pour prouver la terminaison, on cherche une fonction de terminaison f :
  - définie sur base des variables de l'algorithme et à valeur entière naturelle (≥ 0)
  - telle que f décroît strictement suite à l'exécution du corps de la boucle
  - telle que B implique $f > 0$
- Puisque f décroît strictement, elle finira par atteindre 0 et donc à infirmer B.

Exemple : Fibonacci-iter

```
Fibonacci-Iter(n)
    if n ≤ 1
        return n
    else
        pprev = 0
        prev = 1
        for i = 2 to n
            f = prev + pprev
            pprev = prev
            prev = f
        return f
```

Proposition : Si $n \geq 0$, Fibonacci-iter(n) renvoie F_n.

Réécriture, post- et pré-conditions

```
Fibonacci-Iter(n)
    {n ≥ 0} // {P}
    if n ≤ 1
        prev = n
    else
        pprev = 0
        prev = 1
        i = 2
        while (i ≤ n)
            f = prev + pprev
            pprev = prev
            prev = f
            i = i + 1
    {prev == F_n} // {Q}
    return prev
```

Exemple : FIBONACCI-ITER

Analyse de la condition

$\{n \geq 0 \text{ et } n \leq 1\}$
$prev = n$
$\{prev == F_n\}$

correct $(F_0 = 0, F_1 = 1)$

$\{n \geq 0 \text{ et } n > 1\}$
$pprev = 0$
$prev = 1$
$i = 2$
while $(i \leq n)$
    $f = prev + pprev$
    $pprev = prev$
    $prev = f$
    $i = i + 1$
$\{prev == F_n\}$

$I = \{pprev == F_{i-2}, prev == F_{i-1}\}$

Analyse de la boucle

$\{n > 1\}$
$pprev = 0$
$prev = 1$
$i = 2$
$\{pprev == F_{i-2}, prev == F_{i-1}\}$

correct

$\{pprev == F_{i-2}, prev == F_{i-1}, i \leq n\}$
$f = prev + pprev$
$pprev = prev$
$prev = f$
$i = i + 1$
$\{pprev == F_{i-2}, prev == F_{i-1}\}$

correct

$\{pprev == F_{i-2}, prev == F_{i-1}, i == n + 1\}$
$\{prev == F_n\}$

correct

Exemple : Fibonacci-iter

$i = 2$
while $(i \leq n)$
$\quad f = prev + pprev$
$\quad pprev = prev$
$\quad prev = f$
$\quad i = i + 1$

- Fonction de terminaison $f = n - i + 1$:
  - $i \leq n \Rightarrow f = n - i + 1 > 0$
  - $i = i + 1 \Rightarrow f$ diminue à chaque itération
- L'algorithme est donc correct et se termine.

Exemple : tri par insertion

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

- Démontrons informellement que la boucle externe est correcte
- Invariant I : le sous-tableau $A[1..j-1]$ contient les éléments du tableau original $A[1..j-1]$ ordonnés.

Exemple : tri par insertion

$$\textbf{for } j = 2 \textbf{ to } A.length \quad \ldots \quad \Leftrightarrow \quad j = 2 \quad \textbf{while } i \leq A.length \quad \ldots \quad j = j + 1$$

- $P =$"A est un tableau de taille $A.length$",
  $Q =$"Le tableau A est trié",
  $I =$"$A[1..j-1]$ contient les $j-1$ premiers éléments de A triés"

- $\{P\} j = 2 \{I\}$ *(avant la boucle)*
  - $j = 2 \Rightarrow A[1]$ est trivialement ordonné

Exemple : tri par insertion

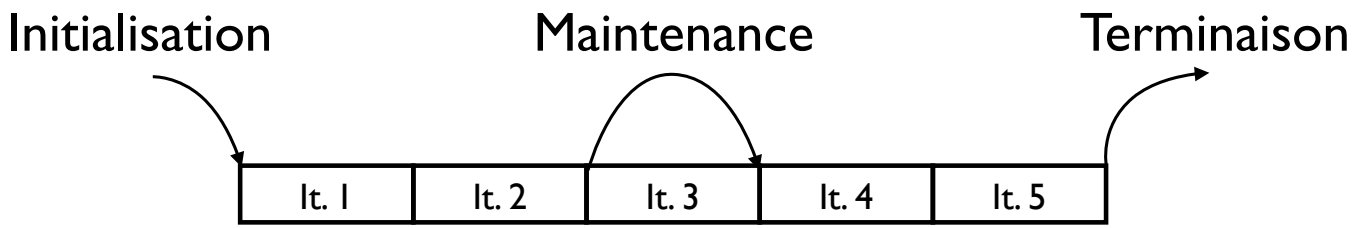
- $\{I \text{ et } j \leq A.length\}$ CORPS $\{I\}$ *(pendant la boucle)*
  - La boucle interne déplace $A[j-1]$, $A[j-2]$, $A[j-3]\ldots$ d'une position vers la droite jusqu'à trouver la bonne position pour *key* ($A[j]$).
  - $A[1\,..\,j]$ contient alors les éléments originaux de $A[1\,..\,j]$ triés.
  - $j = j+1$ rétablit l'invariant
- $\{I \text{ et } j = A.length + 1\}\{Q\}$ *(après la boucle)*
  - Puisque $j = A.length + 1$, l'invariant implique que $A[1\,..\,A.length]$ est ordonné.

- Fonction de terminaison $f = A.length - j + 1$

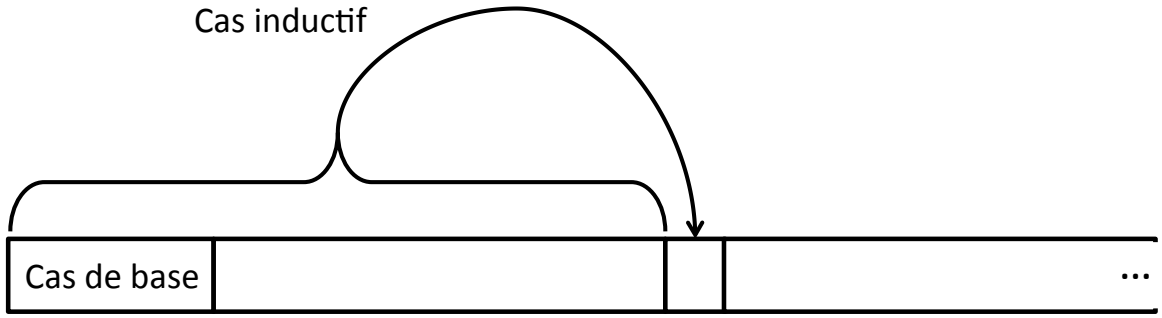
Invariant

- Un invariant peut être difficile à trouver pour certains algorithmes
- En général, l'algorithme découle de l'invariant et pas l'inverse
  - FIBONACCI-ITER : On calcule itérativement F_{i-1}et F_{i-2}
    ($I = \{pprev == F_{i-2}, prev == F_{i-1}\}$)
  - INSERTION-SORT : On ajoute l'élément j aux $j-1$ premiers éléments déjà triés
    ($I =$" $A[1..j-1]$ contient les $j-1$ premiers éléments de A triés")

- La preuve par invariant est basée sur le principe général de preuve par induction qu'on va utiliser aussi pour prouver la correction des algorithmes récursifs



Preuve par induction



- On veut montrer qu'une propriété est vraie pour une série d'instances
- On suppose l'existence d'un ordonnancement des instances
- Cas de base : on montre explicitement que la propriété est vraie pour la ou les premières instances
- Cas inductif : on suppose que la propriété est vraie pour les k premières instances et on montre qu'elle l'est alors aussi pour la $k+1$-ième instance (quel que soit k)
- Par le principe d'induction, la propriété sera vraie pour toutes les instances

Preuve par induction : exemple

Proposition : Pour tout $n \geq 0$, on a

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Démonstration :

- Cas de base : $n = 0 \Rightarrow \sum_{i=1}^{0} i = 0 = \frac{0(0+1)}{2}$
- Cas inductif : Supposons la propriété vraie pour n et montrons qu'elle est vraie pour $n+1$:

$$\begin{aligned} \sum_{i=1}^{n+1} i &= \left(\sum_{i=1}^{n} i\right) + (n+1) = \frac{n(n+1)}{2} + (n+1) \\ &= \frac{(n+1)(n+2)}{2} \end{aligned}$$

- Par induction, le propriété est vraie pour tout n.

Correction d'algorithmes récursifs par induction

- Propriété à montrer : l'algorithme est correct pour une instance quelconque du problème
- Instances du problème ordonnées par "taille" (taille du tableau, nombre de bits, un entier n, etc.)
- Cas de base de l'induction = cas de base de la récursion
- Cas inductif : on suppose que les appels récursifs sont corrects et on en déduit que l'appel courant est correct
- Terminaison : on montre que les appels récursifs se font sur des sous-problèmes (souvent trivial)

Exemple : Fibonacci

```
Fibonacci(n)
1  if n ≤ 1
2        return n
3  return Fibonacci(n − 2) + Fibonacci(n − 1)
```

Proposition : Pour tout n, Fibonacci(n) renvoie F_n.

Démonstration :

- Cas de base : pour $n = 0$, Fibonacci(n) renvoie $F_0 = 0$. Pour $n = 1$, Fibonacci(n) renvoie $F_1 = 1$.
- Cas inductif :
  - Supposons $n \geq 2$ et que pour tout $0 \leq m < n$, Fibonacci(m) renvoie F_m.
  - Pour $n \geq 2$, Fibonacci(n) renvoie

$$\begin{aligned} & \text{Fibonacci}(n-2) + \text{Fibonacci}(n-1) \\ = \; & F_{n-2} + F_{n-1} \text{ (par hypothèse inductive)} \\ = \; & F_n. \end{aligned}$$

Exemple : merge sort

```
MERGE-SORT(A, p, r)
1  if p < r
2        q = ⌊(p+r)/2⌋
3        MERGE-SORT(A, p, q)
4        MERGE-SORT(A, q + 1, r)
5        MERGE(A, p, q, r)
```

Proposition : Pour tout $1 \leq p \leq r \leq A.\mathit{length}$, MERGE-SORT$(A, p, r)$ trie le sous-tableau $A[p \,..\, r]$.

(On supposera que MERGE est correct mais il faudrait le démontrer par un invariant)

Exemple : merge sort

```
MERGE-SORT(A, p, r)
1  if p < r
2        q = ⌊(p+r)/2⌋
3        MERGE-SORT(A, p, q)
4        MERGE-SORT(A, q + 1, r)
5        MERGE(A, p, q, r)
```

Démonstration :

- Cas de base : pour $r - p = 0$, MERGE-SORT(A, p, r) ne modifie pas A et donc $A[p] = A[q]$ est trivialement trié
- Cas inductif :
  - Supposons $r - p > 0$ et que pour tout $1 \leq p' \leq r' \leq A.\mathit{length}$ tels que $r' - p' < r - p$, MERGE-SORT(A, p', r') trie $A[p' \,..\, r']$
  - Les appels MERGE-SORT(A, p, q) et MERGE-SORT$(A, q + 1, r)$ sont corrects par hypothèse inductive (puisque $q - p < r - p$ et $r - q - 1 < r - p$)
  - En supposant MERGE correct, MERGE-SORT(A, p, r) est correct.

Conclusion sur la correction

- Preuves de correction :
  - Algorithmes itératifs : invariant (=induction)
  - Algorithmes récursifs : induction

- Malheureusement, il n'existe pas d'outil automatique pour vérifier la correction (et la terminaison) d'algorithmes
- Dans la suite, on ne présentera des invariants ou des preuves par induction que très sporadiquement lorsque ce sera nécessaire (cas non triviaux)

Plan

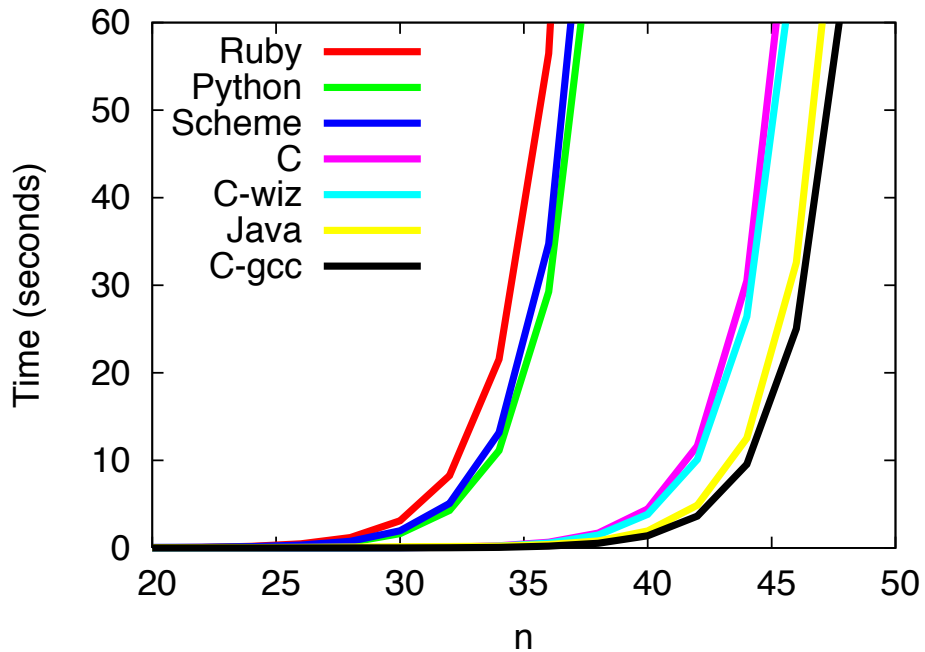
Performance d'un algorithme

- Plusieurs métriques possibles :
  - Longueur du programme (nombre de lignes)
  - Simplicité du code
  - Espace mémoire consommé
  - Temps de calcul
  - ...
- Les temps de calcul sont la plupart du temps utilisés
  - Ils peuvent être quantifiés et sont faciles à comparer
  - Souvent ce qui compte réellement
- Nous étudierons aussi l'espace mémoire consommé par nos algorithmes

Comment mesurer les temps d'exécution ?

Expérimentalement :

- On écrit un programme qui implémente l'algorithme et on l'exécute sur des données
- Problèmes :
  - Les temps de calcul vont dépendre de l'implémentation : CPU, OS, langage, compilateur, charge de la machine, OS, etc.
  - Sur quelles données tester l'algorithme ?



(Carzaniga)

Comment mesurer les temps d'exécution ?

Sur papier :

- Développer un modèle de machine ("Random-access machine", RAM) :
  - Opérations executées les unes après les autres (pas de parallélisme)
  - Opérations de base (addition, affectation, branchement, etc.) prennent un temps constant
  - Appel de sous-routines : temps de l'appel (constant) + temps de l'exécution de la sous-routine (calculé récursivement)
- Calculer les temps de calcul = sommer le temps d'exécution associé à chaque instruction du pseudo-code

- Le temps dépend de l'entrée (l'instance particulière du problème)
- On étudie généralement les temps de calcul en fonction de la "taille" de l'entrée
  - Généralement, le nombre de valeurs pour la décrire
  - Mais ça peut être autre chose (Ex : n pour FIBONACCI)

Analyse du tri par insertion

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

- t_j = nombre de fois que la condition du **while** est testée.
- Temps exécution $T(n)$ (pour un tableau de taille n) donné par :

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j-1)$$

$$+ c_7 \sum_{j=2}^{n}(t_j-1) + c_8(n-1)$$

Différents types de complexité

- Même pour une taille fixée, la complexité peut dépendre de l'instance particulière
- Soit D_n l'ensemble des instances de taille n d'un problème et $T(i_n)$ le temps de calcul pour une instance $i_n \in D_n$.
- Sur quelles instances les performances d'un algorithme devraient être jugées :
  - Cas le plus favorable (best case) : $T(n) = \min\{T(i_n) | i_n \in D_n\}$
  - Cas le plus défavorable (worst case) : $T(n) = \max\{T(i_n) | i_n \in D_n\}$
  - Cas moyen (average case) : $T(n) = \sum_{i_n \in D_n} Pr(i_n) T(i_n)$ où $Pr(i_n)$ est la probabilité de rencontrer i_n
- On se focalise généralement sur le cas le plus défavorable
  - Donne une borne supérieure sur le temps d'exécution.
  - Le meilleur cas n'est pas représentatif et le cas moyen est difficile à calculer.

Analyse du tri par insertion

Meilleur cas :

- le tableau est trié $\Rightarrow t_j = 1$.
- Le temps de calcul devient :

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

- $T(n) = an + b \Rightarrow T(n)$ est une fonction linéaire de n

Analyse du tri par insertion

Pire cas :

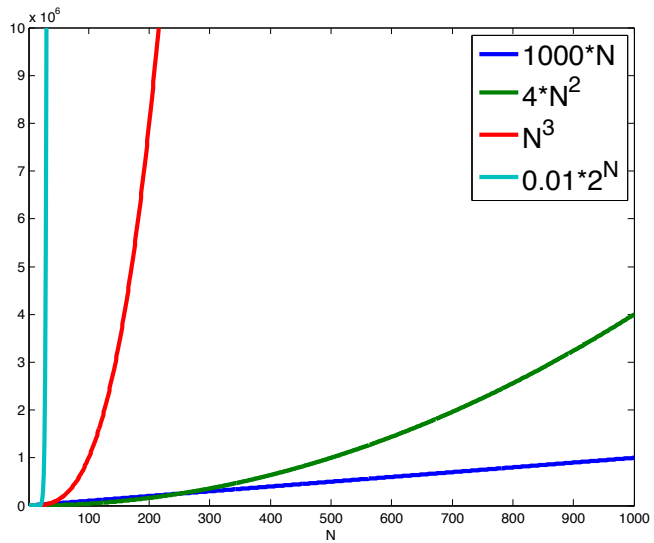
- le tableau est trié par ordre décroissant $\Rightarrow t_j = j$.
- Le temps de calcul devient :

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

- $T(n) = an^2 + bn + c \Rightarrow T(n)$ est une fonction quadratique de n

Analyse asymptotique

- On s'intéresse à la vitesse de croissance ("order of growth") de $T(n)$ lorsque n croît.
  - 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 : Tri par insertion : $T(n) = an^2 + bn + c \rightarrow n^2$.

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	$10ms$	$100s$	~ 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)

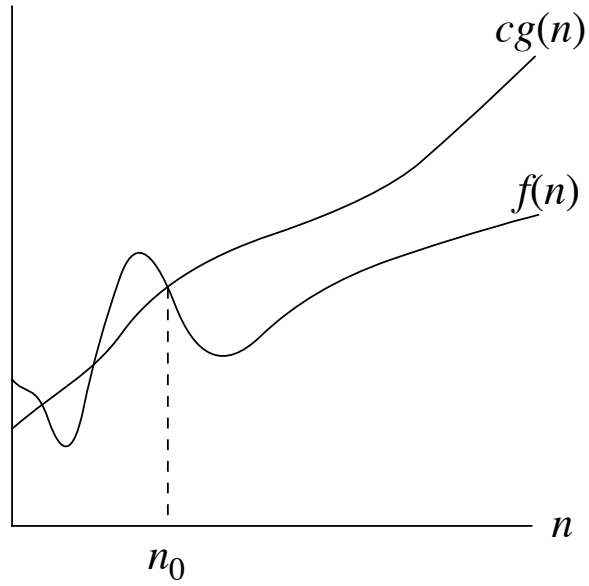
Notations asymptotiques

- Permettent de caractériser le taux de croissance de fonctions $f : \mathbb{N} \to \mathbb{R}^+$

- Trois notations :
  - Grand-O : $f(n) \in O(g(n)) \approx f(n) \leq g(n)$
  - Grand-Omega : $f(n) \in \Omega(g(n)) \approx f(n) \geq g(n)$
  - Grand-Theta : $f(n) \in \Theta(g(n)) \approx f(n) = g(n)$

Notation grand-O

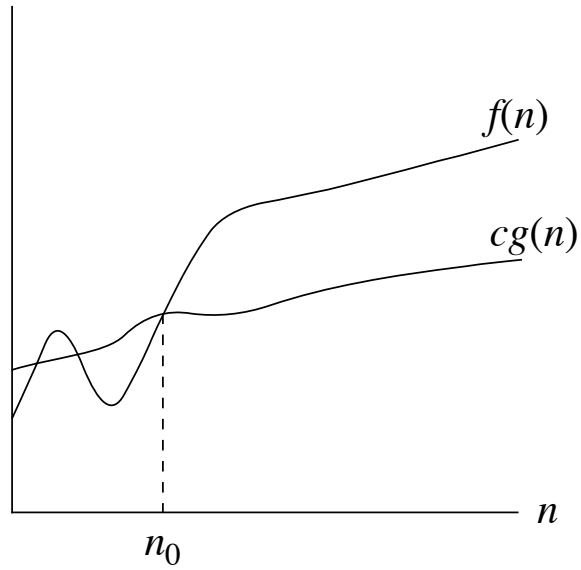
$$O(g(n)) = \{f(n) | \exists c > 0, \exists n_0 \geq 1 \text{ tels que } 0 \leq f(n) \leq cg(n), \forall n \geq n_0\}$$



- $f(n) \in O(g(n)) \Rightarrow g(n)$ est une borne supérieure asymptotique pour $f(n)$.
- Par abus de notation, on écrira aussi : $f(n) = O(g(n))$.

Notation grand-Omega

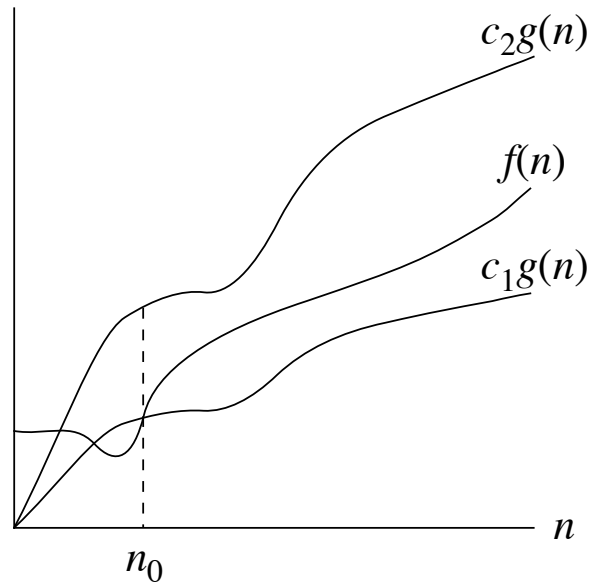
$$\Omega(g(n)) = \{f(n) | \exists c > 0, \exists n_0 \geq 1 \text{ tels que } 0 \leq cg(n) \leq f(n), \forall n \geq n_0\}$$



- $f(n) \in \Omega(g(n)) \Rightarrow g(n)$ est une borne inférieure asymptotique pour $f(n)$.
- Par abus de notation, on écrira aussi : $f(n) = \Omega(g(n))$.

Notation grand-Theta

$$\Theta(g(n)) = \{f(n) | \exists c_1, c_2 > 0, \exists n_0 \geq 1 \text{ tels que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$$



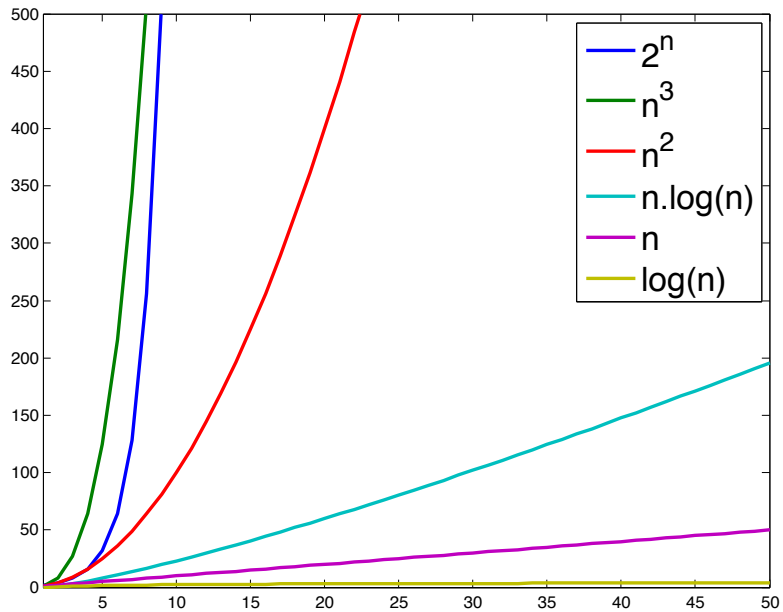
- $f(n) \in \Theta(g(n)) \Rightarrow g(n)$ est une borne serrée (“tight”) asymptotique pour $f(n)$.
- Par abus de notation, on écrira aussi : $f(n) = \Theta(g(n))$.

Exemples

- $3n^5 - 16n + 2 \in O(n^5) ? \in O(n) ? \in O(n^{17}) ?$
- $3n^5 - 16n + 2 \in \Omega(n^5) ? \in \Omega(n) ? \in \Omega(n^{17}) ?$
- $3n^5 - 16n + 2 \in \Theta(n^5) ? \in \Theta(n) ? \in \Theta(n^{17}) ?$
- $2^n + 100n^6 + n \in O(2^n) ? \in \Theta(3^n) ? \in \Omega(n^7) ?$

- Classes de complexité :

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



Quelques propriétés

- $f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in O(f(n))$
- $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ et $f(n) \in \Omega(g(n))$
- $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$

- 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
  $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$

D'autres notations (pour information)

- Equivalence asymptotique : $f(n) \sim g(n)$ ssi $\lim_{n\to\infty} f(n)/g(n) = 1$
  - f et g sont asymptotiquement équivalents.
  - Permet de comparer deux fonctions en prenant en compte la constante
- Petit-o :

$$o(g(n)) = \{f(n) | \forall c > 0, \exists n_0 \geq 1 \text{ tels que } 0 \leq cg(n) \leq f(n), \forall n \geq n_0\}$$

  - f est négligeable devant g asymptotiquement
  - $\approx f(n) < g(n)$
- Petit-ω :

$$\omega(g(n)) = \{f(n) | \forall c > 0, \exists n_0 \geq 1 \text{ tels que } 0 \leq cg(n) \leq f(n), \forall n \geq n_0\}$$

  - f domine g asymptotiquement
  - $\approx f(n) > g(n)$

Complexité d'un algorithme

- On utilise les notations asymptotiques pour caractériser la complexité d'un algorithme.
- Il faut préciser de quelle complexité on parle : générale, au pire cas, au meilleur cas, en moyenne...
- La notation grand-O est de loin la plus utilisée
  - $f(n) \in O(g(n))$ sous-entend généralement que $O(g(n))$ est le plus petit sous-ensemble qui contient $f(n)$ et que $g(n)$ est la plus concise possible
  - Exemple : $n^3 + 100n^2 - n \in O(n^3) = O(n^3 + n^2) \subset O(n^4) \subset O(2^n)$
- Idéalement, les notations O et Ω devraient être limitées au cas où on n'a pas de borne serrée.

Complexité d'un algorithme

Exemples :

- On dira :
  "La complexité au pire cas du tri par insertion est $\Theta(n^2)$"
  plutôt que
  "La complexité au pire cas du tri par insertion est $O(n^2)$"
  ou "La complexité du tri par insertion est $O(n^2)$"
- On dira
  "La complexité au meilleur cas du tri par insertion est $\Theta(n)$"
  plutôt que
  "La complexité au meilleur cas du tri par insertion est $\Omega(n)$"
  ou "La complexité du tri par insertion est $\Omega(n)$"
- Par contre, on dira "La complexité de FIBONACCI est $\Omega(1.4^n)$", car on n'a pas de borne plus précise à ce stade.

Complexité d'un problème

- Les notations asymptotiques servent aussi à caractériser la complexité d'un problème
  - Un problème est $O(g(n))$ s'il existe un algorithme $O(g(n))$ pour le résoudre
  - Un problème est $\Omega(g(n))$ si tout algorithme qui le résoud est forcément $\Omega(g(n))$
  - Un problème est $\Theta(g(n))$ s'il est $O(g(n))$ et $\Omega(g(n))$
- Exemple du problème de tri :
  - Le problème du tri est $O(n \log n)$ (voir plus loin)
  - On peut montrer facilement que le problème du tri est $\Omega(n)$ (voir le transparent suivant)
  - On montrera plus tard que le problème de tri est en fait $\Omega(n \log n)$ et donc qu'il est $\Theta(n \log n)$.
- Exercice : montrez que la recherche du maximum dans un tableau est $\Theta(n)$

Le problème du tri est $\Omega(n)$

Preuve par l'absurde (ou par contraposition) :

- Supposons qu'il existe un algorithme moins que $O(n)$ pour résoudre le problème du tri
- Cet algorithme ne peut pas parcourir tous les éléments du tableau, sinon il serait au moins $O(n)$
- Il y a donc au moins un élément du tableau qui n'est pas vu par cet algorithme
- Il existe donc des instances de tableau qui ne seront pas triées correctement par cet algorithme
- Il n'y a donc pas d'algorithme plus rapide que $O(n)$ pour le tri.

Comment calculer la complexité en pratique ?

Quelques règles simples pour les algorithmes itératifs :

- 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 $O(f(n))$

Comment calculer la 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 i = 1 to n
    for j = 1 to i
        ...
```

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

```
i = 1
while i ≤ n
    ...
    i = 2i
```

Exemple :

PREFIXAVERAGES(X) :

- **Entrée :** tableau X de taille n
- **Sortie :** tableau A de taille n tel que $A[i] = \frac{\sum_{j=1}^{i} X[j]}{i}$

```
PREFIXAVERAGES(X)
1  for i = 1 to X.length
2        a = 0
3        for j = 1 to i
4              a = a + X[j]
5        A[i] = a/i
6  return A
```

Complexité : $\Theta(n^2)$

```
PREFIXAVERAGES2(X)
1  s = 0
2  for i = 1 to X.length
3        s = s + X[i]
4        A[i] = s/i
5  return A
```

Complexité : $\Theta(n)$

Cas plus compliqué

- Pour des algorithmes plus complexes, appliquez les règles précédentes peut facilement mener à surestimer la complexité.

- Approche scientifique :
  - On détecte une *opération abstraite* au cœur de l'algorithme
  - On développe un *modèle des entrées* de l'algorithme
  - On détermine une *expression analytique* pour le nombre d'exécutions $T(N)$ de l'opération pour une taille d'entrée N.
  - On fait l'hypothèse que le coût de l'algorithme est $\approx aT(N)$ où a est une constante et on en déduit la complexité en notation asymptotique

- Exemple du problème de tri : opération abstraite est la comparaison

Complexité d'algorithmes récursifs

- La complexité d'algorithmes récursifs mène généralement à une équation de récurrence
- La résolution de cette équation n'est généralement pas triviale
- On verra dans la section suivante différentes techniques génériques de résolution de récurrence

Factorial et Fibonacci

```
Factorial(n)
1  if n == 0
2        return 1
3  return n · Factorial(n − 1)
```

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

$$\Rightarrow T(n) \in \Theta(n)$$

```
Fib(n)
1  if n ≤ 1
2        return n
3  return Fib(n − 2) + Fib(n − 1)
```

$$\begin{aligned} T(0) &= c_0, T(1) = c_0 \\ T(n) &= T(n-1) + T(n-2) + c_1 \end{aligned}$$

$$\Rightarrow T(n) \in \Omega(1.4^n)$$

Analyse du tri par fusion

```
MERGE-SORT(A, p, r)
1  if p < r
2        q = ⌊(p+r)/2⌋
3        MERGE-SORT(A, p, q)
4        MERGE-SORT(A, q + 1, r)
5        MERGE(A, p, q, r)
```

- Récurrence :

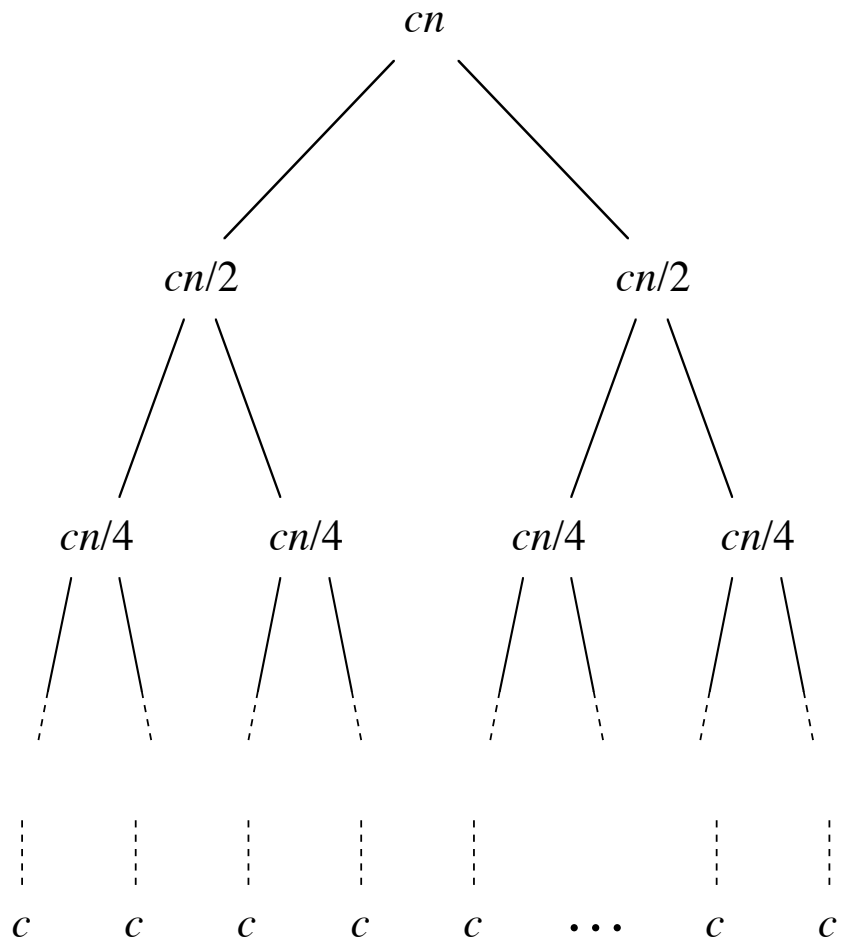
$$\begin{aligned} T(1) &= c_1 \\ T(n) &= 2T(n/2) + c_2 n + c_3 \end{aligned} \qquad \begin{aligned} T(1) &= \Theta(1) \\ T(n) &= 2T(n/2) + \Theta(n) \end{aligned}$$

Analyse du tri par fusion

- Simplifions la récurrence en :

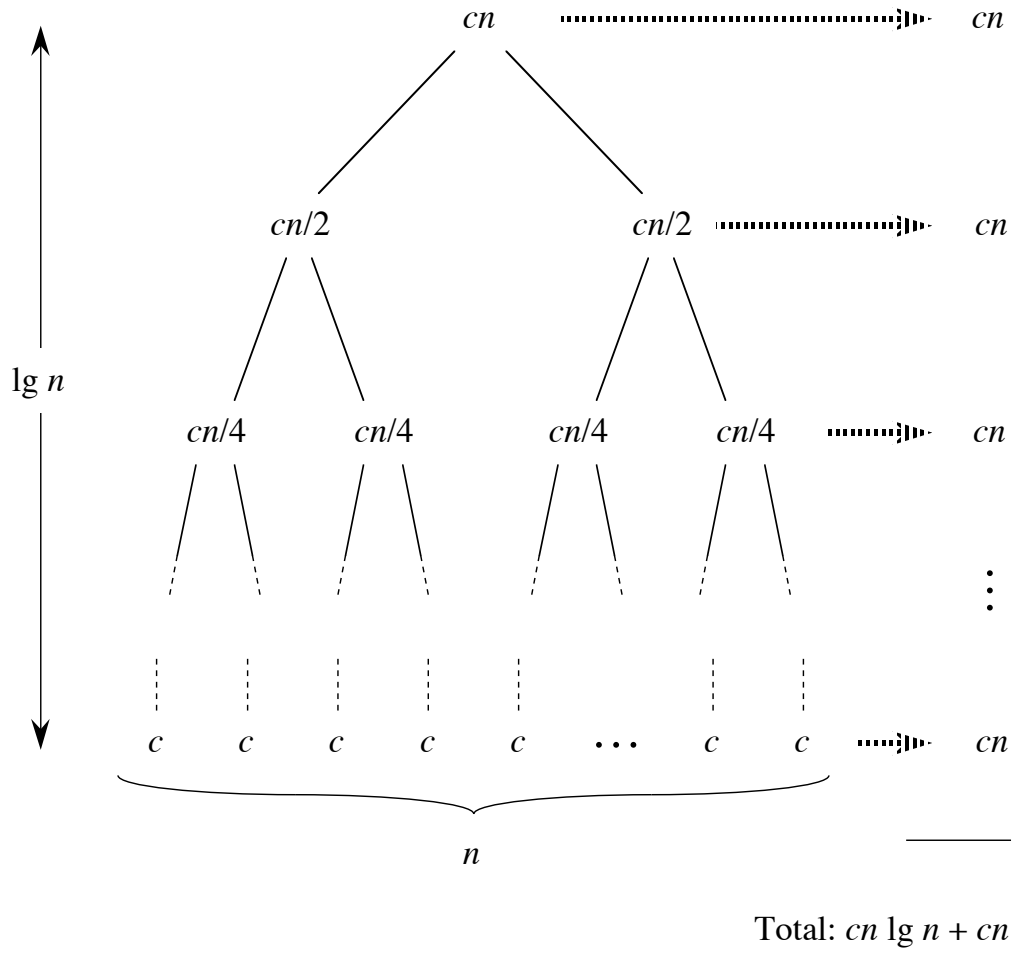
$$\begin{aligned} T(1) &= c \\ T(n) &= 2T(n/2) + cn \end{aligned}$$

- On peut représenter la récurrence par un arbre de récursion
- La complexité est la somme du coût de chaque noeud



Analyse du tri par fusion

- Chaque niveau a un coût cn
- En supposant que n est une puissance de 2, il y a $\log_2 n + 1$ niveaux
- Le coût total est $cn \log_2 n + cn \in \Theta(n \log n)$



Remarques

Limitations de l'analyse asymptotique

- Les facteurs constants ont de l'importance pour des problèmes de petite taille
  - Le tri par insertion est plus rapide que le tri par fusion pour n petit
- Deux algorithmes de même complexité (grand-O) peuvent avoir des propriétés très différentes
  - Le tri par insertion est en pratique beaucoup plus efficace que le tri par sélection sur des tableaux presque triés

Complexité en espace

- S'étudie de la même manière, avec les mêmes notations
- Elle est bornée par la complexité en temps (pourquoi ?)

Plan

Sommations et récurrences

- Les analyses de complexité font très souvent intervenir des sommations et des récurrences
- Dans cette section, on va voir quelques techniques génériques pour trouver une solution analytique à des sommations et des récurrences
- Une *solution analytique* est une expression mathématique qui peut être évaluée à l'aide d'un nombre constant d'opérations de base (addition, multiplication, exponentiation, etc.).

Sommations

Définition : Soit une suite $x_i (i \in \mathbb{Z})$. La sommation $\sum_{i=a}^{b} x_i$ pour $a, b \in \mathbb{Z}$ est définie récursivement par :

- $\sum_{i=a}^{b} x_i = 0$ si $b < a$ *(cas de base)*
- $\sum_{i=a}^{b} x_i = x_a$ si $b = a$ *(cas de base)*
- $\sum_{i=a}^{b} x_i = \left(\sum_{i=a}^{b-1} x_i\right) + x_b$ si $b > a$ *(cas inductif)*

Définition : Soit une suite de réels x_i, $i \in \mathbb{N}$, la *série* de *terme général* x_i est la suite de *sommes partielles*

$$\sum_{i=0}^{n} x_i \ (n \in \mathbb{N}).$$

Etant donné une série, on notera S_n la n-ème somme partielle $\sum_{i=0}^{n} x_i$.
La suite des sommes partielles peut être définie récursivement :

- $S_0 = x_0$
- $S_n = S_{n-1} + x_n$ pour $n > 0$

Preuve d'une solution analytique

Une solution analytique se prouve généralement facilement par induction.

Exemple : Série géométrique :

Théorème : Pour tous $n \geq 1$ et $z \neq 1$, on a

$$\sum_{i=0}^{n-1} z^i = \frac{1-z^n}{1-z}.$$

Démonstration : La preuve fonctionne par induction.
$P(n) =$ " $\sum_{i=0}^{n-1} z^i = \frac{1-z^n}{1-z}$ "
Cas de base $(n = 1)$: $P(1)$ est vérifié
Cas inductif $(n > 1)$: Si $P(n)$ est vérifié, on peut écrire :

$$\sum_{i=0}^{n} z^i = \sum_{i=0}^{n-1} z^i + z^n = \frac{1-z^n}{1-z} + z^n = \frac{1-z^n+z^n-z^{n+1}}{1-z} = \frac{1-z^{n+1}}{1-z}$$

□

(Exercice : montrer que $\sum_{i=0}^{n} i^2 = \frac{(2n+1)(n+1)n}{6}$ *)*

Sommes infinies

Définition : $\sum_{i=0}^{\infty} z_i = \lim_{n\to\infty} \sum_{i=0}^{n} z_i$.

Théorème : Si $|z| < 1$, alors $\sum_{i=0}^{\infty} z^i = \frac{1}{1-z}$.

Démonstration :

$$\begin{aligned}
\sum_{i=0}^{\infty} z^i &= \lim_{n\to\infty} \sum_{i=0}^{n} z^i \\
&= \lim_{n\to\infty} \frac{1-z^{n+1}}{1-z} \\
&= \frac{1}{1-z}.
\end{aligned}$$

□

Trouver une solution analytique

Si prouver une solution analytique est aisé, imaginer cette solution l'est moins.

Différentes techniques génériques existent :

- Dériver cette solution de la solution analytique d'une autre série (par exemple par dérivation ou intégration),
- Méthode de perturbation,
- Par identification paramétrique.
- ...

Variantes des séries géométriques

Théorème : Pour tous $n \geq 0$ et $z \neq 1$, on a

$$\sum_{i=0}^{n} iz^i = \frac{z - (n+1)z^{n+1} + nz^{n+2}}{(1-z)^2}.$$

Démonstration : On a

$$\sum_{i=0}^{n} iz^i = z \cdot \sum_{i=0}^{n} iz^{i-1} = z \cdot \left(\frac{d}{dz} \sum_{i=0}^{n} z^i \right) = z \cdot \left(\frac{d}{dz} \frac{1-z^{n+1}}{1-z} \right).$$

En développant, on obtient

$$\begin{aligned} & z \cdot \left(\frac{d}{dz} \frac{1-z^{n+1}}{1-z} \right) \\ = \quad & z \cdot \left(\frac{-(n+1)z^n(1-z) - (-1)(1-z^{n+1})}{(1-z)^2} \right) \end{aligned}$$

$$
\begin{aligned}
&= z \cdot \left(\frac{-(n+1)z^n + (n+1)z^{n+1} + 1 - z^{n+1}}{(1-z)^2} \right) \\
&= z \cdot \left(\frac{1 - (n+1)z^n + nz^{n+1}}{(1-z)^2} \right) \\
&= \frac{z - (n+1)z^{n+1} + nz^{n+2}}{(1-z)^2}.
\end{aligned}
$$

□

Corollaire : Si $|z| < 1$, alors $\sum_{i=0}^{\infty} iz^i = \frac{z}{(1-z)^2}$.

Autre variante : En intégrant les deux côtés de $\sum_{i=0}^{\infty} z^i = \frac{1}{1-z}$ (de 0 à x), on peut obtenir :

$$\sum_{j=1}^{\infty} \frac{x^j}{j} = -\ln(1-x).$$

Méthode de perturbation

Soit S_n la n-ème somme partielle de la série de terme général x_i. Par définition, on a

$$S_n + x_{n+1} = x_0 + \sum_{i=1}^{n+1} x_k \ (= S_{n+1})$$

Si on peut exprimer le membre de droite comme une fonction de S_n, on peut obtenir une solution analytique en résolvant l'équation pour S_n.

Exemple : Pour la série géométrique $S_n = \sum_{i=0}^{n-1} z^i$:

$$S_{n+1} = S_n + z^n = z^0 + \sum_{i=1}^{n} z^i = 1 + z \sum_{i=0}^{n-1} z^i = 1 + zS_n$$

D'où, on tire immédiatement :

$$S_n = \frac{1 - z^n}{1 - z}$$

Un autre exemple

Problème[1] : Dériver une solution analytique de $S_n = \sum_{k=0}^{n} k2^k$.

Solution : Par la méthode de perturbation :

$$\begin{aligned}
S_n + (n+1)2^{n+1} &= 0 \cdot 2^0 + \sum_{k=1}^{n+1} k2^k = \sum_{k=1}^{n+1} k2^k \\
&= \sum_{k=0}^{n} (k+1)2^{k+1} \\
&= \sum_{k=0}^{n} k2^{k+1} + \sum_{k=0}^{n} 2^{k+1} \\
&= 2\sum_{k=0}^{n} k2^k + 2\sum_{k=0}^{n} 2^k \\
&= 2S_n + 2(2^{n+1} - 1) \\
\Rightarrow S_n &= (n-1)2^{n+1} + 2
\end{aligned}$$

1. Cette somme apparaît dans l'analyse du tri par tas (voir partie 3).

Par identification

On fait une hypothèse sur la forme de la solution et on identifie les paramètres en prenant quelques valeurs.

Exemple : $\sum_{i=1}^{n} i^2 = \frac{(2n+1)(n+1)n}{6}$.

- Supposer que la somme est un polynôme de degré 3 (car somme$\sim$intégration)

$$\sum_{i=1}^{n} i^2 = an^3 + bn^2 + cn + d$$

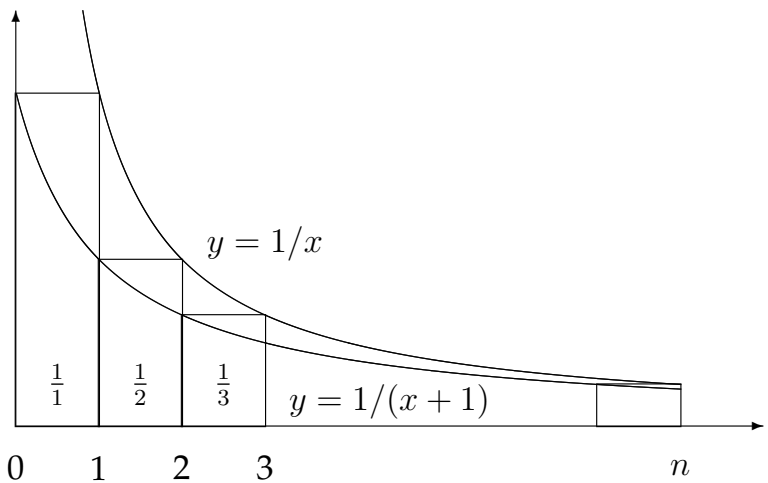
- Identifier les constantes a, b, c, d à partir de quelques valeurs de la somme
- Prouver sa validité par induction (!)

Borner une série par intégration

- Certaines séries n'ont pas de solution analytique (connue).
- Exemple : La *série harmonique* H_n :

$$H_n = \frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n}$$

- Dans ce cas, des bornes inférieures et supérieures peuvent cependant être déterminées par intégration



$$\begin{aligned}
\int_0^n \frac{1}{x+1}\,dx &\leq H_n \leq 1 + \int_1^n \frac{1}{x}\,dx \\
[\ln(x+1)]_0^n &\leq H_n \leq 1 + [\ln x]_1^n \\
\ln(n+1) &\leq H_n \leq 1 + \ln(n)
\end{aligned}$$

$$\Rightarrow H_n \in \Theta(\ln(n))$$

(On peut montrer que $H_n \sim \ln(n)$)

Sommes doubles

Généralement, il suffit d'évaluer la somme intérieure et puis la somme extérieure.
Exercice : montrez que $\sum_{n=0}^{\infty}(y^n \sum_{i=0}^{n} x^i) = \frac{1}{(1-y)(1-xy)}$

Quand la somme intérieure n'a pas de solution analytique, échanger les deux sommes peut aider.
Exemple :

$$\begin{aligned}
\sum_{k=1}^{n} H_k &= \sum_{k=1}^{n}\sum_{j=1}^{k}\frac{1}{j} \\
&= \sum_{j=1}^{n}\sum_{k=j}^{n}\frac{1}{j} \\
&= \dots \\
&= (n+1)H_n - n
\end{aligned}$$

		j						
		1	2	3	4	5	...	n
k	1	1						
	2	1	1/2					
	3	1	1/2	1/3				
	4	1	1/2	1/3	1/4			
		...						
	n	1	1/2		...			1/n

Remarque sur les produits

Les mêmes techniques peuvent être utilisées pour calculer des produits en utilisant le logarithme :

$$\prod f(n) = \exp\left(\ln\left(\prod f(n)\right)\right) = exp\left(\sum \ln f(n)\right).$$

Permet de borner $n! = 1 \cdot 2 \cdot 3 \ldots (n-1) \cdot n$. Par la méthode d'intégration, on a :

$$n\ln(n) - n + 1 \leq \sum_{i=1}^{n} \ln(i) \leq (n+1)\ln(n+1) - n.$$

En prenant l'exponentielle :

$$\frac{n^n}{e^{n-1}} \leq n! \leq \frac{(n+1)^{(n+1)}}{e^n}$$

Stirling's formula :

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

Plan

1. Correction d'algorithmes
2. Complexité algorithmique
3. Résolution de sommations et de récurrences
    - Sommation
    - Récurrences

Récurrences

- La complexité d'algorithmes récursifs est souvent calculable à partir d'équations récurrentes
- Exemples
  - Tri par fusion

$$\begin{aligned} T(1) &= 0 \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1 \text{ pour } n > 1 \end{aligned}$$

  - Fibonacci :

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n-1) + T(n-2) + 2 \text{ pour } n > 1 \end{aligned}$$

  - Tour de Hanoï (cf. INFO0947)

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n-1) + 1 \text{ pour } n > 1 \end{aligned}$$

Classification des récurrences

Forme générale : $u_n = f(\{u_0, u_1, \ldots, u_{n-1}\}, n)$

Une récurrence peut être :

- *linéaire* si f est une combinaison linéaire à coefficients *constants* ou *variables*.
  - $u_n = 3u_{n-1} + 4u_{n-2}$, $u_n = n * u_{n-2} + 1$
- *polynomiale* si f est un polynôme.
  - $b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}$
- *d'ordre k* si f dépend de $u_{n-1}, \ldots, u_{n-k}$.
- *complète* si f dépend de $u_{n-1}, \ldots, u_0$.
- *de type "diviser pour régner"* si f dépend de $u_{n/a}$, avec $a \in \mathbb{N}$ constant.
  - $u_n = 2u_{n/2}$
- *homogène* si f ne dépend que des u_i.
  - $u_n = 2u_{n/2} + u_{n/4}$
- *non homogène* si elle est de la forme $u_n = f(\{u_p | p < n\}) + g(n)$.
  - $u_n = 2u_{n-1} + n$

Techniques de résolution de récurrences

On souhaite obtenir une solution analytique à une récurrence.

Plusieurs approches possibles :

- “Deviner-et-vérifier”
- “Plug-and-chug” (téléscopage)
- Arbres de récursion
- Equations caractéristique (linéaire, voir MATH0491)
- Master theorem (diviser-pour-régner)
- Changement de variable
- ...

Méthode “Deviner-et-Vérifier”

Principes :

1. Calculer les quelques premières valeurs de T_n ;
2. Deviner une solution analytique ;
3. Démontrer qu'elle est correcte, par exemple par induction.

Application :

- $T_n = 2T_{n-1} + 1$ (tours de Hanoï) :

n	1	2	3	4	5	6	$\cdots$
T_n	1	3	7	15	31	63	$\cdots$

  $\Rightarrow$ On devine $T_n = 2^n - 1$

- On doit démontrer (par induction) que la solution est correcte.

Preuve d'une solution par induction

Théorème : $T_n = 2^n - 1$ satisfait la récurrence :

$$\begin{aligned} T_1 &= 1 \\ T_n &= 2T_{n-1} + 1 \text{ pour } n \geq 2. \end{aligned}$$

Démonstration : Par induction sur n. $P(n) =$ " $T_n = 2^n - 1$".
Cas de base : $P(1)$ est vrai car $T_1 = 1 = 2^1 - 1$.
Cas inductif : Montrons que $T_n = 2^n - 1$ (pour $n \geq 2$) est vrai dès que $T_{n-1} = 2^{n-1} - 1$ est vrai :

$$\begin{aligned} T_n &= 2T_{n-1} + 1 \\ &= 2(2^{n-1} - 1) + 1 \\ &= 2^n - 1. \end{aligned}$$

□

Méthode “Plug-and-Chug” (force brute)

(aussi appelée téléscopage ou méthode des facteurs sommants)

1. “Plug” (appliquer l’équation récurrente) et “Chug” (simplifier)

$$\begin{aligned} T_n &= 1 + 2T_{n-1} \\ &= 1 + 2(1 + 2T_{n-2}) \\ &= 1 + 2 + 4T_{n-2} \\ &= 1 + 2 + 4(1 + 2T_{n-3}) \\ &= 1 + 2 + 4 + 8T_{n-3} \\ &= \dots \end{aligned}$$

Remarque : Il faut simplifier *avec modération*.

2. Identifier et vérifier un “pattern”
   - Identification :

$$T_n = 1 + 2 + 4 + \cdots + 2^{i-1} + 2^i T_{n-i}$$

   - Vérification en développant une étape supplémentaire :

$$\begin{aligned} T_n &= 1 + 2 + 4 + \cdots + 2^{i-1} + 2^i(1 + 2T_{n-(i+1)}) \\ &= 1 + 2 + 4 + \cdots + 2^{i-1} + 2^i + 2^{i+1} T_{n-(i+1)} \end{aligned}$$

3. Exprimer le $n^{\text{ème}}$ terme en fonction des termes précédents
   En posant $i = n - 1$, on obtient

$$\begin{aligned} T_n &= 1 + 2 + 4 + \cdots + 2^{n-2} + 2^{n-1} T_1 \\ &= 1 + 2 + 4 + \cdots + 2^{n-2} + 2^{n-1} \end{aligned}$$

4. Trouver une solution analytique pour le $n^{\text{ème}}$ terme

$$\begin{aligned} T_n &= 1 + 2 + 4 + \cdots + 2^{n-2} + 2^{n-1} \\ &= \sum_{i=0}^{n-1} 2^i \\ &= \frac{1 - 2^n}{1 - 2} \\ &= 2^n - 1 \end{aligned}$$

Tri par fusion

Appliquons le “plug-and-chug” à la récurrence du tri par fusion *dans le cas où* $n = 2^k$:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1 = 2T(n/2) + n - 1 \text{ pour } n > 1 \end{aligned}$$

Tri par fusion

Appliquons le "plug-and-chug" à la récurrence du tri par fusion *dans le cas où* $n = 2^k$:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1 = 2T(n/2) + n - 1 \text{ pour } n > 1 \end{aligned}$$

- Pattern :

$$\begin{aligned} T(n) &= 2^i T(n/2^i) + (n - 2^{i-1}) + (n - 2^{i-2}) + \ldots + (n - 2^0) \\ &= 2^i T(n/2^i) + i \cdot n - 2^i + 1 \end{aligned}$$

- En posant $i = k$ et en utilisant $k = \log_2 n$:

$$\begin{aligned} T(n) &= 2^k T(n/2^k) + k \cdot n - 2^k + 1 \\ &= nT(1) + n \log_2 n - n + 1 \\ &= n \log_2 n - n + 1 \end{aligned}$$

Arbres de récursion

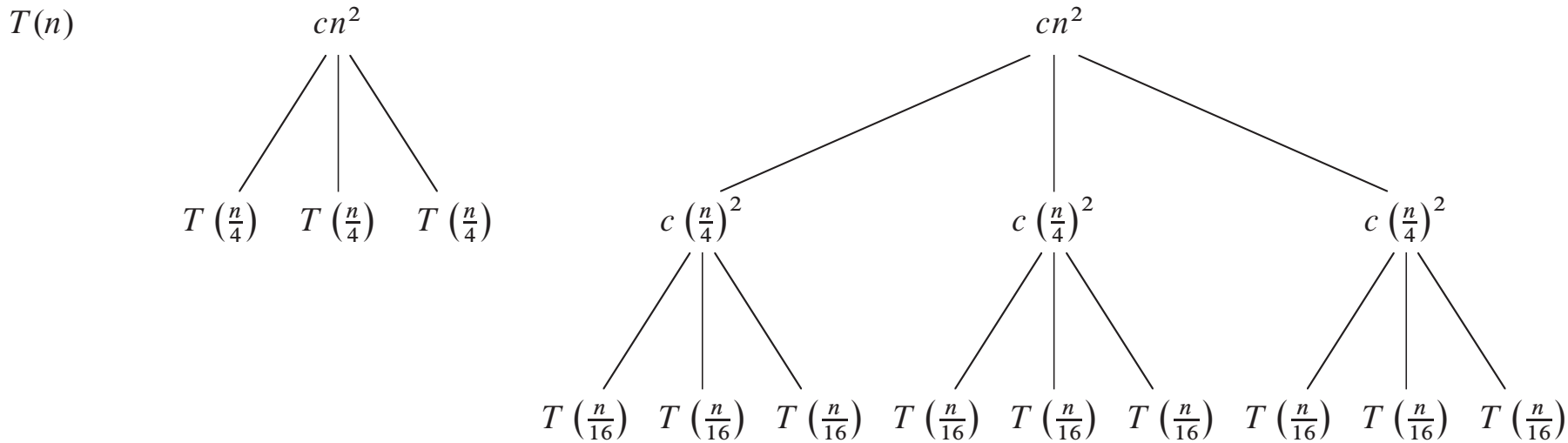
Approche graphique pour *deviner* une solution analytique (ou une borne asymptotique) à une récurrence.

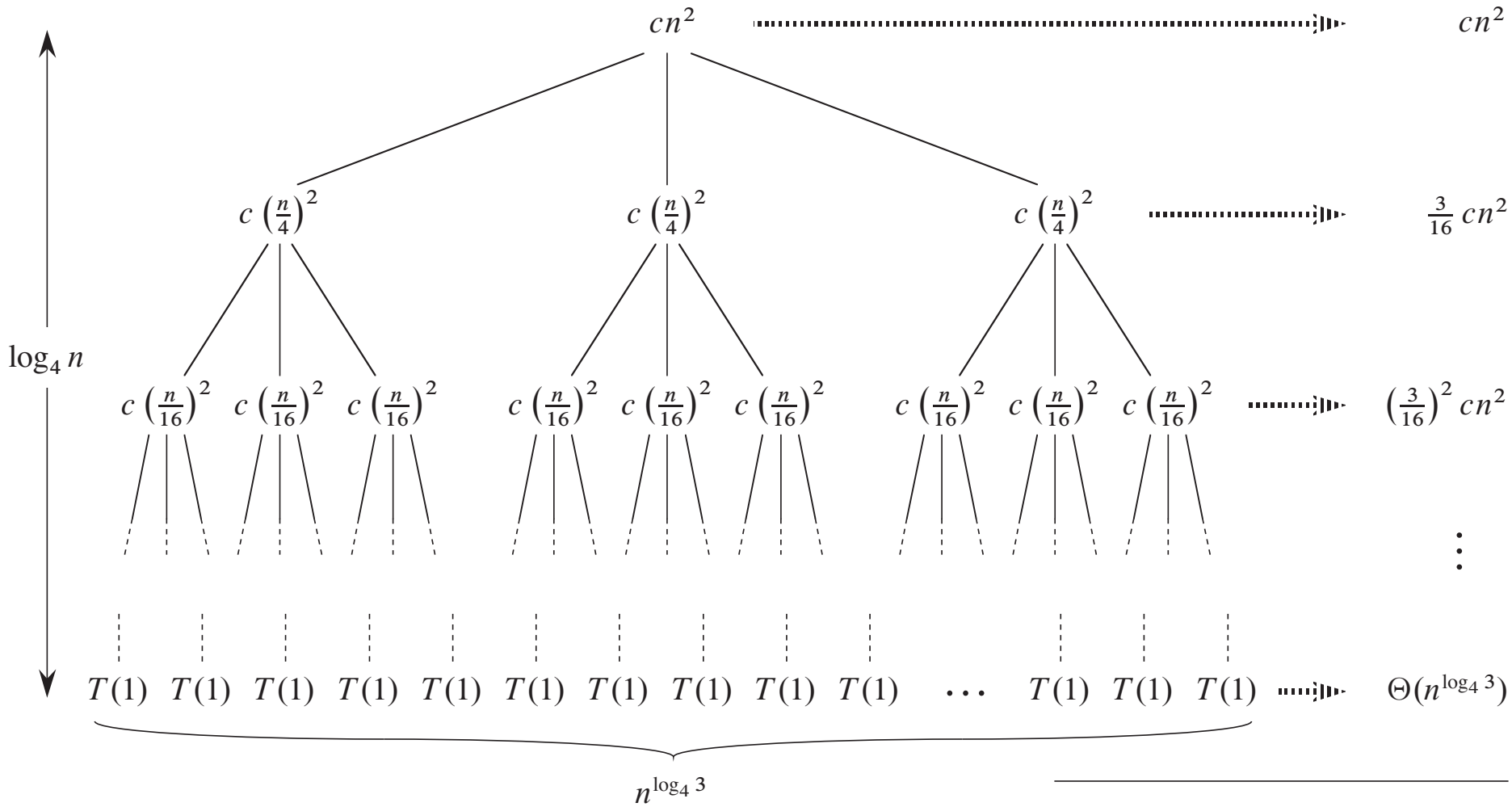
La solution devra toujours être démontrée ensuite (par induction).

Illustration sur la récurrence suivante :

- $T(1) = a$
- $T(n) = 3T(n/4) + cn^2$ (Pour $n > 1$)

(Introduction to algorithms, Cormen et al.)


$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$
cn^2
$c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$


cn^2
cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$\frac{3}{16}\,cn^2$
$\log_4 n$
$c\left(\frac{n}{16}\right)^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$T(1)$
$\Theta(n^{\log_4 3})$
$n^{\log_4 3}$

- Le coût total est la somme du coût de chaque niveau de l'arbre :

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \ldots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + an^{\log_4 3} \\ &= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + an^{\log_4 3} \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + an^{\log_4 3} \\ &= \frac{1}{1-(3/16)} cn^2 + an^{\log_4 3} \\ &\in O(n^2) \end{aligned}$$

(à vérifier par induction)

- Comme le coût de la racine est cn^2, on a aussi $T(n) \in \Omega(n^2)$ et donc $T(n) \in \Theta(n^2)$.

Preuve d'une borne supérieure

Théorème : Soit la récurrence :

- $T(1) = a$,
- $T(n) = 3T(n/4) + cn^2$ (Pour $n > 1$).

On a $T(n) \in O(n^2)$.

Préparation de la démonstration : Soit $P(n) =$ "$T(n) \leq dn^2$". L'idée est de trouver un d tel que $P(n)$ peut se montrer par induction forte.
Cas de base : $P(1)$ est vrai si $a \leq d \cdot 1^2 = d$.
Cas inductif : Supposons $P(1), P(4), \ldots, P(n/4)$ vérifiés et trouvons une contrainte sur d telle que $P(n)$ le soit aussi :

$$\begin{aligned} T(n) &\leq 3T(n/4) + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\leq dn^2, \end{aligned}$$

La dernière étape est valide dès que $d \geq (16/13)c$. Le théorème est donc vérifié pour autant que $d \geq \max\{(16/13)c, a\}$.

Preuve d'une borne supérieure

Ré-écriture de la preuve :

Démonstration : Soit une constante d telle que $d \geq \max\{(16/13)c, a\}$.
Montrons par induction forte que $P(n) =$ "$T(n) \leq dn^2$" est vrai pour tout $n \geq 1$ (qui implique $T(n) \in O(n)$).
Cas de base : $P(1)$ est vrai puisque $d \geq \max\{(16/13)c, a\} \geq a$.
Cas inductif : Supposons $P(1), P(4), \ldots, P(n/4)$ vérifiés et montrons que $P(n)$ l'est aussi :

$$\begin{aligned} T(n) &\leq 3T(n/4) + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\leq dn^2, \end{aligned}$$

où la dernière étape découle de $d \geq max((16/13)c, a)$.

□

Synthèse

Trois approches empiriques pour trouver une solution analytique :

- "Deviner-et-vérifier"
- "Plug-and-Chug"
- Arbres de récursion

Ces approches sont génériques mais

- il n'est pas toujours aisé de trouver le pattern ;
- il faut pouvoir résoudre la somme obtenue ;
- la solution doit être vérifiée par induction.

Il existe des méthodes plus systématiques pour résoudre des récurrences particulières

- Récurrences linéaires d'ordre $k \geq 1$ à coefficients constants : solution analytique exacte (voir MATH0491)
- Récurrences "diviser-pour-régner" (borne asymptotique uniquement)

Récurrence générale “diviser-pour-régner”

Définition : Un récurrence “diviser-pour-régner” est une récurrence de la forme :

$$T_n = \sum_{i=1}^{k} a_i T(b_i n) + g(n),$$

où $a_1, \ldots, a_k$ sont des constantes positives, $b_1, \ldots, b_k$ sont des constantes comprises entre 0 et 1 et $g(n)$ est une fonction non négative.

Exemple : $k = 1$, $a_1 = 2$, $b_1 = 1/2$ et $g(n) = n - 1$ correspond au tri par fusion

Sous certaines conditions, il est possible de trouver des bornes asymptotiques sur les récurrences de ce type.

Master theorem

Théorème : Soit la récurrence suivante :

$$\begin{cases} T(n) = c & \text{si } n < d \\ T(n) = aT(\frac{n}{b}) + f(n) & \text{si } n \geq d \end{cases}$$

où $d \geq 1$ est une constantes entière, $a > 0$, $c > 0$ et $b > 1$ sont des constantes réelles, et $f(n)$ est une fonction positive pour $n \geq d$.

1. Si $f(n) \in O(n^{\log_b a - \epsilon})$ pour un $\epsilon > 0$, alors $T(n) \in \Theta(n^{\log_b a})$
2. Si $f(n) \in \Theta(n^{\log_b a})$, alors $T(n) \in \Theta(n^{\log_b a} \log n)$.
3. Si $f(n) \in \Omega(n^{\log_b a + \epsilon})$ pour un $\epsilon > 0$ et si $af(\frac{n}{b}) \leq \delta f(n)$ pour un $\delta < 1$, alors $T(n) \in \Theta(f(n))$.

(Introduction to algorithms, Cormen et al.)

NB : les bornes ne dépendent pas de c et d. $\frac{n}{b}$ peut être interprété soit comme $\lfloor \frac{n}{b} \rfloor$, soit comme $\lceil \frac{n}{b} \rceil$.

Exemple d'application

- Soit la récurrence suivante :

$$T(n) = 7T(n/2) + \Theta(n^2).$$

  (Méthode de Strassen pour la multiplication de matrice)
- $T(n)$ satisfait aux conditions du théorème avec $a = 7$, $b = 2$.
- $\log_b a = \log_2 7 = 2.807... \Rightarrow f(n) \in O(n^{\log_2 7 - \epsilon})$ avec $\epsilon = 0.8$.
- Par le premier cas du théorème, on a :

$$T(n) = \Theta(n^{\log_b a}) = O(n^{2.807...}).$$

Changement de variables

Un changement de variables permet parfois de transformer une récurrence “diviser pour régner” en une récurrence linéaire.

- Soit la récurrence du transparent précédent :

$$T(n) = 7T(n/2) + \Theta(n^2)$$

- En posant : $n = 2^m$ et $S(m) = T(2^m)$, on obtient :

$$S(m) = T(2^m) = 7T(2^{m-1}) + \Theta((2^m)^2) = 7S(m-1) + \Theta(4^m)$$

Changement de variables

Un changement de variable permet de résoudre des récurrences qui semblent a priori complexes.

- Considérons la récurrence suivante :

$$T(n) = 2T(\sqrt{n}) + \log n$$

- Posons $m = \log n$. On a :

$$T(2^m) = 2T(2^{m/2}) + m.$$

- Soit $S(m) = T(2^m)$. On a :

$$S(m) = 2S(m/2) + m \Rightarrow S(m) \in \Theta(m \log m).$$

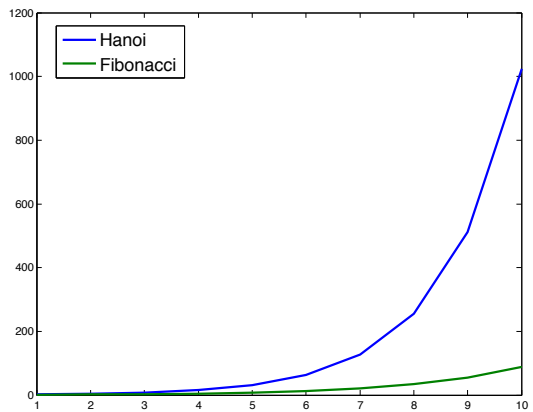
- Finalement :

$$T(n) = T(2^m) = S(m) \in O(m \log m) = O(\log n \log \log n).$$

Comparaisons de récurrences : linéaires versus “d-p-r”

	Récurrence	Solution
Tours de Hanoï	$T_n = 2T_{n-1} + 1$	$T_n \sim 2^n$
Tours de Hanoï 2	$T_n = 2T_{n-1} + n$	$T_n \sim 2 \cdot 2^n$
Algo rapide	$T_n = 2T_{n/2} + 1$	$T_n \sim n$
Tri par fusion	$T_n = 2T_{n/2} + n - 1$	$T_n \sim n \log n$
Fibonacci	$T_n = T_{n-1} + T_{n-2}$	$T_n \sim (1.618\ldots)^{n+1}/\sqrt{5}$

- Récurrences “Diviser pour régner” généralement polynomiales
- Récurrences linéaires généralement exponentielles
- Générer des sous-problèmes petits est beaucoup plus important que de réduire la complexité du terme non homogène
  - Tri par fusion et Fibanocci sont exponentiellement plus rapides que les tours de Hanoï



Comparaisons de récurrences : nombre de sous-problèmes

Récurrences linéaires :

$$T_n = 2T_{n-1} + 1 \quad \Rightarrow \quad T_n = \Theta(2^n)$$
$$T_n = 3T_{n-1} + 1 \quad \Rightarrow \quad T_n = \Theta(3^n)$$

Augmentation exponentielle des temps de calcul quand on passe de 2 à 3 sous-problèmes.

Récurrence “diviser-pour-régner” :

$$\begin{aligned} T_1 &= 0 \\ T_n &= aT_{n/2} + n - 1 \end{aligned}$$

Par le master théorème, on a :

$$T_n = \begin{cases} \Theta(n) & \text{pour } a < 2 \\ \Theta(n \log_2 n) & \text{pour } a = 2 \\ \Theta(n^{\log_2 a}) & \text{pour } a > 2. \end{cases}$$

La solution est complètement différente entre $a = 1.99$ et $a = 2.01$.

Ce qu'on a vu

- Correction d'algorithmes itératifs (par invariant) et récursifs (par induction)
- Notions de complexité algorithmique
- Notations asymptotiques
- Calcul de complexité d'algorithmes itératifs et récursifs
- Plusieurs techniques de résolution de sommations et de récurrences