Complexité

Plan

Machines de Türing, calculabilité et décidabilité.

Classes de complexité traditionnelles et aléatoires

Hiérarchie polynomiale

Introduction aux preuves probabilistes et aux th. PCP.

Cours 1 et 2

1. Explication simple de la complexité, de la calculabilité et de la décidabilité
2. Relation entre les nombres et les chaînes de caractère
3. Passage à Türing

Cours 3 et 4

1. Exo MT,
2. Décidabilité et calculabilité
3. Langages récursifs primitifs
4. Réduction
5. MT non déterministe

Plan cours 4 et 5

Les classes de complexité traditionnelles : P, NP, NP-Complet

Réduction de problèmes NP-Complets : SAT, SSP, …,

Calcul de complexité

Plan cours 6 : complexité aléatoire (Kolmogorov)

Notion de grandeur :
Problème du voyageur de commerce (PVC) :
20 villes : 19 !/2 = 9.10^{16}
30 villes : 29 !/2 = 3.10^{29}
40 villes : 39 !/2 = 10^{46}
Soit un ordinateur de diamètre 1mm, on en place tout autour de l'équateur. :
$6378.4*10^3*10^3*2*PI=4.10^{10}$
Chacun peut calculer 10^3 chemins par secondes : 4.10^{13}chemins calculés par seconde
Pour 20 villes il faut 2.10^3 s pour résoudre le problème (33mn)
Pour 22 villes il faut presque 10 jours pour résoudre le problème
Pour 24 villes il faut 13 ans
Pour 30 villes 1 milliard d'années
Or on a des problèmes à résoudre avec l'équivalent de 100 000 villes.

Pour 30 villes il faut 10^{16}s soit

Avant d'essayer de calculer la complexité d'un problème il faut

1. S'assurer que le problème possède une solution
2. S'assurer que l'on peut mettre en place un algorithme permettant de la résoudre

Exemples d'algorithmes : le crible d'Eratosthène pour déterminer si un nombre est premier

Construire la liste des nombres premiers jusqu'à N

Ecrire tous les entiers jusqu'à N puis éliminer les multiples de 2, puis les multiples de 3 parmi ceux restant, …

Théorème de Fermat : déterminer si pour tout entier n>2, il existe des entiers positifs a, b, c tels que $a^n+b^n=c^n$

Conjecture de Golbach : tout nombre pair peut s'écrire comme la somme de deux nombres premiers.

Alan Türing (1912-1954)

Dénombrabilité

Tout algorithme qui pourra être implémenté sur un ordinateur devra fonctionner sur des ensembles dénombrables. On pourra ainsi traiter des entiers, des couples d'entiers, des suites, des caractères, …, à condition que les ensembles définis soient énumérables.

Si un ensemble est énumérable il existe une bijection entre cet ensemble et $\mathbb{N}$ donc nous raisonnerons sur $\mathbb{N}$ pour simplifier les écritures.

Tout ensemble fini est dénombrable. {0, 1, 2, .., n}, {paul, marie, …, jean}, $\{0,1, \ldots,n\}^p$.

$\mathbb{N}$ est dénombrable, $\mathbb{N}^p$ est dénombrable si p est fini.

L'union ou le produit fini d'ensembles dénombrables est dénombrable.

$\mathbb{R}$ n'est pas dénombrable.

Démonstration (Diagonale de Cantor)

Raisonnons sur l'intervalle [0,1[. Les éléments de cet intervalle sont des suites de chiffres : 0,13456 ; 0,74938475 ; … on peut donc identifier [0,1[comme étant l'ensemble des suites (finies et infinies) de la forme : $0,a_1a_2a_3a_4a_{5\ldots}$

Si cet ensemble est dénombrable alors on peut classer les différentes suites dans un tableau.

1	X_{11}	X_{12}	X_{13}	…	X_{1p}	…
2	X_{21}	X_{22}	X_{23}	…	X_{2p}	…

3	X_{31}	X_{32}	X_{33}	…	X_{3p}	…
4	X_{41}	X_{42}	X_{43}	…	X_{4p}	…
…						
k-1	X_{k-11}	X_{k-12}	X_{k-13}	…	X_{k-1p}	…
k	X_{k1}	X_{k2}	X_{k3}	…	X_{kp}	…
…						

Si on considère la suite composée de tous les termes diagonaux auxquels on ajoute 1 avec la règle suivante : 9+1 = 0

1	$\mathbf{X_{11}+1}$	X_{12}	X_{13}	…	X_{1p}	…
2	X_{21}	$\mathbf{X_{22}+1}$	X_{23}	…	X_{2p}	…
3	X_{31}	X_{32}	$\mathbf{X_{33}+1}$	…	X_{3p}	…
4	X_{41}	X_{42}	X_{43}	…	X_{4p}	…
…						
k-1	X_{k-11}	X_{k-12}	X_{k-13}	…	X_{k-1p}	…
k	X_{k1}	X_{k2}	X_{k3}	…	X_{kp}	…
…						

Cette suite n'appartient pas au tableau puisqu'elle diffère de chacune des suites déjà présentes par au moins un élément (l'élément diagonal).

Donc $\mathbb{R}$ n'est pas dénombrable.

Passage des chaînes aux nombres entiers

Les chaînes de caractères que l'on appellera mot sur un alphabet constituent un ensemble infini dénombrable, on peut donc associer à chacune d'entre elle un entier. Tout calcul sur une chaîne peut donc se ramener à un calcul sur des entiers.

Passage des nombres entiers aux chaînes

Pour calculer sur des entiers, il faut leur donner une représentation sous forme de chaîne dans un langage (binaire par exemple), donc les calculs sur des entiers se ramènent à des calculs sur des chaînes.

Savoir par exemple si un nombre est pair revient à savoir si la représentation de ce nombre correspond à la représentation d'un nombre pair.

Les ensembles de chaînes définissent des langages et tout problème se ramène donc à savoir si un mot, codant le problème, appartient ou non à un langage.

Langages

Définitions :

1. **Alphabet** : ensemble fini de caractères Σ, on ajoute le caractère vide ε

2. **Mot** : suite finie de caractères générée à partir d'un alphabet, l'ensemble des mots générés à partir d'un alphabet Σ est noté Σ^*

Décidabilité

Exemple de problèmes indécidables

1. Le $10^{ème}$ problème de Hilbert (1862-1943) : existe-t-il un algorithme pour résoudre les équations Diophantines ? Equation Diophantine : déterminer si une équation de la forme P=0 où P est un polynôme à coefficients entiers possède des solutions entières.
2. Déterminer si un programme calcule bien une fonction donnée

Ces problèmes ne sont pas décidables (démonstration en 1970 par Yuri Matijasevic pour le $10^{ème}$ problème de Hilbert) c'est-à-dire qu'il n'existe pas d'algorithme qui pour chaque occurrence du problème nous indique si il possède une solution.

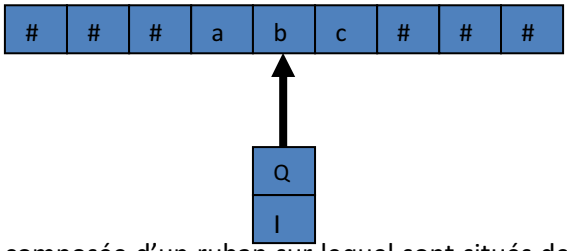
Définition : soit P un prédicat (une propriété) applicable à un entier n, P est décidable si et seulement si il existe une méthode (un algorithme) permettant de dire au bout d'un temps fini (nombre d'étapes de calcul) si P(n) est vrai ou faux.

Calculabilité

Définition : soit f une fonction de N dans N, f est calculable si et seulement si il existe une méthode de calcul (un algorithme) permettant d'obtenir f(n) pour tout entier n.

Machine de Türing

Physiquement une machine de Türing ressemble à

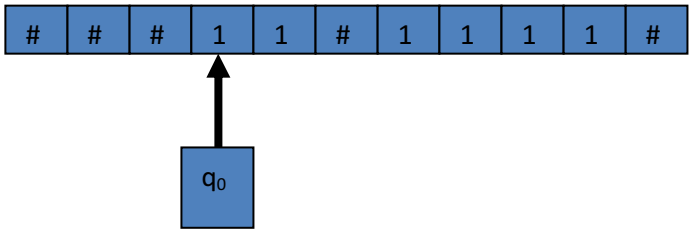


Elle est composée d'un ruban sur lequel sont situés des caractères de l'alphabet (a,b, c ici) ainsi que l'espace (#). Une tête de lecture (la flèche), lit sur le ruban d'entrée un caractère et exécute l'instruction I à partir de l'état Q.

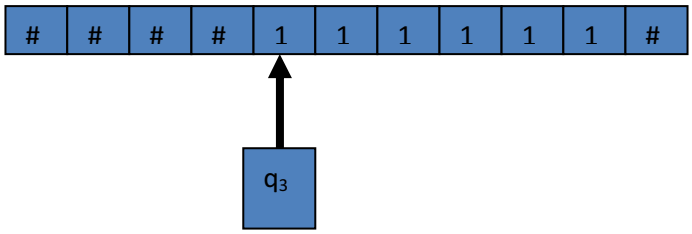
Une instruction I est un quintuple ***(q,a,a',q',d)*** se traduisant par : si la machine est dans l'état ***q*** et qu'elle lit ***a*** sur le ruban alors elle le remplace par ***a'*** et va dans l'état ***q'***, puis se déplace dans la direction ***d*** (droite ou gauche).

> Le déroulement d'une machine de Türing consiste à partir d'une configuration initiale, d'exécuter tant que possible les instructions et de fournir comme résultat le contenu du ruban.

Exemple : soit le jeu d'instructions {(q_0,1,1,q_0,d), (q_0,#,1,q_1,g), (q_1,1,1,q_1,g), (q_1,#,#,q_2,d), (q_2,1,#,q_3,d), (q_2,#,#,q_3,d)} et la configuration suivante



On obtiendra au terme de l'exécution la configuration suivante



Exercice : donnez l'ensemble des configurations intermédiaires.

Quelle fonction remplit cette machine de Türing ?

Configuration

(q,xa,by) $\xrightarrow{M}$(q,xu,vy) avec (u,v)=(ε, ab') si (q,b,b',q',g) est une transition et (u,v)=(ab', ε) si (q,b,b',q',d)

est une transition, avec a,b, b' ϵ Σ, x, y ϵ Σ^*

On note $\xrightarrow{M^*}$ la fermeture transitive de $\xrightarrow{M}$. C'est-à-dire que : c $\xrightarrow{M^*}$ c'si et seulement si il existe une suite

de configurations c_0, c_1, …, c_n telle que $c \xrightarrow{M} c_0 \xrightarrow{M} c_1 \xrightarrow{M} \ldots \xrightarrow{M} c_n \xrightarrow{M} c'$

Soit ω une valeur dite indéfinie. On associe à M (machine de Türing) la machine $\bar{M}$ définie par

$$\bar{M}(x) = \left\{ \begin{array}{c} y, si\ x\epsilon\Sigma^* \text{et il existe } (q_i, y', y) \text{avec} (q_0, \epsilon, x) \xrightarrow{M^*} (q_i, y', y) \text{et } q_i \text{état terminal} \\ \omega, sinon \end{array} \right\}$$

On dit que la machine de Türing est fiable si pour aucun $x\epsilon\Sigma^*$, $\bar{M}(x) = \omega$

Exercice : reprendre l'additionneur avec la notation des configurations.

Définition de la calculabilité avec les machines de Türing

Une fonction $f: \Sigma^* \rightarrow \Sigma^*$est dite calculable s'il existe une machine de Türing M sur Σ telle que $f(x) = \bar{M}(x)$ pour tout $x\epsilon\Sigma^*$

Remarque : une fonction peut être définie par un ensemble de doublets (x_i,$f(x_i)$).

Remarque : Il existe des fonctions non calculables

Démonstration :

1. L'ensemble des fonctions calculables est dénombrables. En effet, l'ensemble G_n des fonctions représentables par une machine de Türing à n états est fini (nombre fini de caractères, nombre fini d'états, donc nombre fini de transitions entre les états, donc nombre fini de machines de Türing). On peut donc dénombrer chaque ensemble G_i et par suite les fonctions calculables en dénombrant G_1, G_2, … G_n. On note g_0, g_1, …, g_i, … les fonctions calculables.
2. De la même manière le nombre de mots de longueur n construits à partir d'un alphabet quelconque est fini. On peut donc dénombrer l'ensemble des mots générés à partir d'un alphabet en commençant par dénombrer les mots de longueur 1, puis 2, …, puis n. On note x_0, …, x_i, … les mots.
3. On obtient ainsi le tableau suivant

	g_0	g_1	g_2	…	g_i	…
x_0	$g_0(x_0)$	$g_1(x_0)$	$g_2(x_0)$	…	$g_i(x_0)$	…
x_1	$g_0(x_1)$	$g_1(x_1)$	$g_2(x_1)$	…	$g_i(x_1)$	…
x_2	$g_0(x_2)$	$g_1(x_2)$	$g_2(x_2)$	…	$g_i(x_2)$	…
…						
x_i	$g_0(x_i)$	$g_1(x_i)$	$g_2(x_i)$	…	$g_i(x_i)$	…
…						

4. Soit la fonction d :$x_i \rightarrow g_i(x_i)$ (diagonale du tableau).
5. Soit d' une fonction telle que pour tout y d'(y)≠d(y)
6. La fonction d' n'est pas calculable. Si elle l'était elle serait dans le tableau à une certaine colonne j et donc d'=g_j donc d'(x_j)=g_j(x_j)=d(x_j) d'où la contradiction.
7. Donc toutes les fonctions ne sont pas calculables.
8. Par ailleurs on montre que la fonction d n'est pas non plus calculable.

Définition de la décidabilité avec les machines de Türing

On s'intéresse ici à deux mots particuliers du langage noté **vrai** et **faux.** Et aux fonctions qui a tout mot de Σ^* associent vrai ou faux.

On dit que la reconnaissance d'un langage est décidable (ou que L est décidable) si il existe une fonction calculable f telle que : $x \epsilon L \; ssi \; f(x) = vrai$

Il existe des langages dont la reconnaissance est indécidable.

Exemple de problème indécidable :

1. faire un programme qui teste si un autre se termine. Equivalence avec Türing : le problème de l'arrêt d'une machine de Türing est indécidable ? (est-ce que pour tout $x \epsilon \Sigma^*$ M(x) s'arrête).

Supposons que ce programme existe

Prog test_terminaison(u) : = vrai si u() se termine, faux sinon

Prog rec() = while test_terminaison(rec) on boucle

Rec se termine si rec ne se termine pas d'où contradiction.

2. Problème de correspondance de post : soit deux listes de même longueur $A=a_1,\ldots,a_k$ et $B=b_1,\ldots,b_k$ de mots de Σ^*. $\forall i$ (a_i,b_i) est appelé un couple correspondant. Une instance (deux listes A et B) du PCP a une solution si : $\exists$ i_1, i_2, …, i_m tel que $a_{i1}a_{i2}\ldots a_{im}=b_{i1}b_{i2}\ldots b_{im}$. Ce problème est indécidable.
   Exemple : $\Sigma = \{0,1\}$

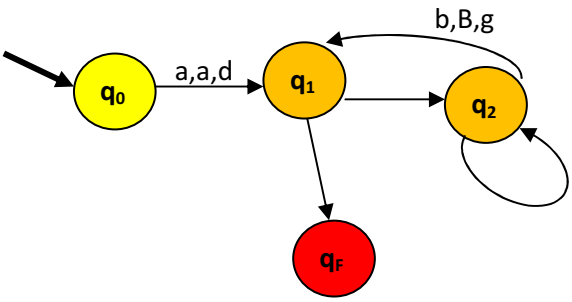
	A	B
1	1	111
2	10111	10
3	10	0

   Il y a une solution pour ce problème : $a_2a_1a_1a_3=b_2b_1b_1b_3$

	A	B
1	10	101
2	011	11
3	101	011

   Il n'y a pas de solution pour ce problème.
3. Considérons des dominos carrés dont chaque côté est coloré. On suppose que l'on peut prendre des dominos en nombre arbitraire dans un type fini donné à l'avance. Peut-on recouvrir toute surface avec des carrés compatibles (dont les côtés qui se touchent sont de même couleur) ? Peut-on recouvrir le plan avec des carrés compatibles ?

Machine de Türing et automates



Réduction de problème

Réduire un problème A à un problème B veut dire que : pour toute instance i du problème A, on peut construire une instance i' du problème B telle que la réponse à la question de B sur i' est oui si et seulement si la réponse à la question de A sur i est oui.

Un algorithme pour B fournit un algorithme pour A.

On utilise la réduction dans plusieurs cas, par exemple pour prouver l'indécidabilité d'un problème, on peut prouver qu'il se réduit en un problème indécidable.

Langages récursifs

Un langage L est récursivement énumérable (RE ou semi-décidable) si il est accepté par une MT M (si M(m) = vrai alors m $\in$ L, si M(m)=non m $\notin$ L, si M(m) = ω m $\notin$ L.

Un langage est récursif s'il est reconnu par une MT qui s'arrête.

Remarque : si L est récursif, $\bar{L}$ est récursif. Si L et $\bar{L}$ sont RE alors L est récursif.

TODO : complexité en moyenne

Fonctions récursives primitives (RP): on s'intéresse aux fonctions de $\mathbb{N}^k$dans $\mathbb{N}$ et on construit des fonctions récursives primitives de proche en proche en partant des trois fonctions suivantes

1. Fonction nulle
2. Fonction Successeur : Succ(t) = t+1
3. Les projections : $(x_1,x_2,\ldots,x_k) \rightarrow x_i$

Et en itérant les deux constructions suivantes :

1. La composition de fonctions : si g_1, g_2, …, g_k sont RP sur $\mathbb{N}^n$et si h est RP sur $\mathbb{N}^k$ alors f=h(g_1, g_2, …, g_k) est RP sur $\mathbb{N}^n$.
2. La définition récursive d'une fonction : si g est RP sur $\mathbb{N}^n$et h RP sur $\mathbb{N} \times \mathbb{N} \times \mathbb{N}^k$, on définit une nouvelle fonction RP sur $\mathbb{N}^{n+1}$par :
    a. f(0,y)=g(y)
    b. f(Succ(x),y) =h(x,f(x,y),y)

Les fonctions RP se programment dans tout langage de programmation à l'aide d'une simple instruction itérative for

```
Function f(x,y)
Z=g(y)
For i=0 to x-1 do
        Z=h(i,z,y)
End
Return(z)
```

Exemples :

Prédécesseur d'un entier

Pred(0)=0
Pred(Succ(x)) = x

Ici n=0, g=fonction identiquement nulle, h(x,y)=x : projection sur la première composante.

Somme de deux entiers

Somme(0,y)=y
Somme(Succ(x),y)=Succ(somme(x,y))
Ici n=1, g(y)=y,h(x,y,z)=Succ(z). La somme de fonctions RP est RP.

Produit de deux entiers

Produit(0,y)=0
Produit(Succ(x),y)=Somme(y,produit(x,y))
Le produit de fonctions RP est RP.

Signe d'un entier

Fonction valant 0 pour x=0 et 1 pour x>0

Sg(0)=0
Sg(Succ(x))=1
Différence tronquée
Vaut y-x si x<y, 0 sinon
Soustrait(0,y)=y
Soustrait(Succ(x),y)=Pred(Soustrait(x,y))
Valeur absolue
Abs(x,y)=Soutrait(x,y)+Soustrait(y,x)
Maximum
Max(x,y)=x+Soustrait(x,y)
Minimum
Min(x,y)=Soustrait(max(x,y),x+y)

Les machines de Türing non déterministes

Jusqu'à présent, les machines de Türing étaient déterministes. C'est-à-dire qu'à un caractère sur le ruban et un état de la machine de Türing ne correspondait qu'une seule transition.

Il existe des machines de Türing non déterministes (MTND) pour lesquelles il existe plusieurs transitions possibles à partir d'un caractère et d'un état.

Un mot est accepté par une MTND s'il existe une exécution de la MT qui donne vrai pour ce mot.

Un langage est reconnu par une MTND si

- pour tout mot du langage il existe une exécution qui donne vrai.
- pour tout mot qui n'appartient pas au langage, aucune exécution ne donne vrai.

On peut convertir toute MTND en une MTD. Le nombre d'états et le temps d'exécution peuvent augmenter de manière importante lors de cette conversion.

Illustration : pour le problème du voyageur de commerce, on peut imaginer une MTND dans laquelle chaque état correspond à un nombre de villes placées sur le chemin et qu'entre deux états, toutes les transitions sont possibles. Avec une telle MTND on peut potentiellement trouver le bon chemin en n itérations.

Les classes de complexité

Classe P

la complexité en temps est le nombre d'étapes de calcul effectuées par une machine de Türing avant d'entrer dans un état terminal.

$T_M(x)=\max\{m : \exists\, x\epsilon\Sigma^*, |x|=n$ et le temps de calcul de M sur x nécessite m pas de calcul$\}$

Max : on est dans le pire cas pour une entrée de taille n

Exemple : pour chercher une valeur dans un tableau : le meilleur cas : position 1, le pire cas : position n.

Soit L_M le langage reconnu par le MTD M. (L_M={ $x\epsilon\Sigma^*$,M accepte x})

La classe de complexité P en temps est définie de la manière suivante : P = {L : ∃M MTD polynomiale : $L=L_M$}

Pour la classe de complexité en espace, on mesure la taille du ruban nécessaire à l'exécution de l'algorithme.

Classe NP

Dans la classe NP, on rassemble les algorithmes qui peuvent être vérifiés en un temps polynomial. C'est-à-dire qu'à l'affirmation : « voici la solution du problème » on doit être capable de répondre « oui c'est la bonne réponse » en un temps polynomial.

On ne prend pas en compte ici le temps qu'il a fallu (qui peut être non polynomial) pour expliciter la solution proposée.

On peut également voir la classe NP comme étant la classe des algorithmes modélisables par une machine de Türing non déterministe dont l'exécution se ferrait en un temps polynomial.

$T_M(x)$=max{1,{m : ∃ $x\epsilon L_M$,|x|=n et le temps de calcul de M sur x nécessite m pas de calcul}}

1 est la dans le cas où il n'existe pas de mot de taille n accepté par M.

La classe NP est définie de la manière suivante : NP={L :∃M | M MTND polynomiale telle que L_M=L}

P est inclus dans NP. On suppose que P est différent de NP mais cela n'a encore jamais été prouvé.

Thèse de Church-Türing : tout modèle raisonnable (algo) est équivalent au modèle de la MT équivalente.

Trouver la complexité d'un problème c'est trouver la complexité du meilleur algorithme qui permette de le résoudre.

Pour prouver la NP-Complétude d'un problème, on prouve qu'il peut être réduit en un problème NP-Complet.

L'analyse de la complexité d'un algorithme consiste à déterminer un fonction f :N→R^+ qui à un paramètre *n* dépendant de la donnée soumise à l'algorithme (en général, *n* est la taille de cette donnée), associe le coût *f*(*n*) (exprimé en unités arbitraires de temps ou d'espace) de l'exécution de algorithme pour la donnée.

Il y a trois types de complexité

La complexité dans le pire des cas : calcul du coût dans le pire des cas,
La complexité en moyenne : on calcule le coût pour chaque donnée possible puis on divise la somme de ces coûts par le nombre de données différentes,
La complexité dans le meilleur des cas : on calcule le coût en se plaçant dans le meilleur des cas.

Le type de complexité dépend de l'application (par exemple : réseau téléphonique, commande de freinage, ...)

NP et MTND :

un langage est reconnu par une MTND en un temps f(n) si pour tout mot du langage il existe une exécution de la MT qui donne vrai en au plus f(n) instructions.

Un langage appartient à la classe NP si f(n) est polynomial.

Résolution de récurrence linéaire (calcul de limite de suites utilisé pour le calcul de la complexité d'algorithmes récursifs).

Illustration : Hanoï, tri par sélection

Suite : $c(n+1) = aC(n) + f(n), a \geq 1$

Si a=1 : $c(n) = c(1) + \sum_{k=1}^{n-1} f(k)$ si $f = O(n^p)$ alors $c(n) = O(n^p)$

Si a>=2 :

si $\frac{f(k)}{a^k}$ converge alors $c(n) = O(a^n)$

Si $f(n) = O(a^n)$ alors $c(n) = O(na^n)$

Si $f(n) = O(b^n)$ avec b>a alors $c(n) = O(b^n)$

Résolution de récurrence « diviser pour régner »

Prenons le cas d'une division en deux parties égales

$$c(n) = aC\left(\frac{n}{2}\right) + bC\left(\frac{n}{2}\right) + f(n), a \geq 0, b \geq 0, a + b \geq 1$$

Soit $\alpha = \log(a+b)$ et f une fonction croissante.

Si $f(n) = O(n^\beta)$ avec $\beta < \alpha$ alors $c(n) = O(n^\alpha)$

Si $f(n) = O(n^\alpha)$ alors $c(n) = O(n^\alpha \ln(n))$

Si $f(n) = O(n^\beta)$ avec $\beta > \alpha$ alors $c(n) = O(n^\beta)$