Introduction à la complexité

Table des matières

[Glossaire formule math 2](#_Toc115346707)

[Avant-propos 2](#_Toc115346708)

[Exemples 3](#_Toc115346709)

[Mesure de la complexité 4](#_Toc115346710)

[Premières mesures de l’efficacité 4](#_Toc115346711)

[Exemple1 4](#_Toc115346712)

[Exemple 2 4](#_Toc115346713)

[Exemple 3 5](#_Toc115346714)

[Exercice 1 6](#_Toc115346715)

[Exercice 2 8](#_Toc115346716)

[Mesure de la complexité 9](#_Toc115346717)

[Démarche générique de la mesure de la complexité : 9](#_Toc115346718)

[Complexité algorithmique et problèmes mathématiques 9](#_Toc115346719)

[Méthodes de calcul de la complexité 10](#_Toc115346720)

[Appel séquentiel 10](#_Toc115346721)

[Appel conditionnel 10](#_Toc115346722)

[Appel itératif 10](#_Toc115346723)

[Exercices de mesure de la complexité 12](#_Toc115346724)

[Classes de complexité 19](#_Toc115346725)

[Majoration asymptotique 19](#_Toc115346726)

[Exemple graphique sur Geogebra 20](#_Toc115346727)

[A quoi ça sert ? 21](#_Toc115346728)

[Exemple graphique avec Geogebra 22](#_Toc115346729)

[Minoration asymptotique 23](#_Toc115346730)

[Exemple graphique sur Geogebra 23](#_Toc115346731)

[Signification de l’appartenance à et à 24](#_Toc115346732)

[Les grandes familles de classe complexité 25](#_Toc115346733)

# Glossaire formule math

Ouvrir une nouvelle formule : Alt+=

* Nouvelle formule : Alt+=
* Indice : a\_1+espace :
* Exposant : a^1+espace :
* Quelque soit : \forall+espace :
* Somme : \sum\_(a)^(b)+espace x :
* Signe multiplier : \times+espace :
* Double implication : \Leftrightarrow +espace:
* Il existe : \exists+espace :
* Omega : \Omega+espace :
* Theta : \Theta+espace :

# Avant-propos

Pour pouvoir comparer des programmes, il faut avoir une métrique indépendante du matériel qui les exécutera. En effet, deux matériels différents risquent d’influencer la mesure permettant la comparaison et donc fausser le résultat.

De plus, un algorithme est une suite d’instructions manipulant des objets mathématiques (nombre, des tableaux, des fonctions…)

Et un programme est l’implémentation dans un langage informatique d’un algorithme. Il est donc contraint par des limites matérielles.

Il est donc nécessaire, pour comparer des programmes, de comparer les algorithmes qu’ils sont sensés implémenter pour éviter toutes considérations matérielles qui pourraient influencer les résultats.

On peut utiliser deux critères pour évaluer « l’efficacité » d’un algorithme :

* Le nombre d’instructions qu’il contient
* La quantité de mémoire nécessaire à son exécution

Grâce au principe d’équivalence « espace/temps », on peut « troquer » de la mémoire utilisée pour des instructions. C-à-d qu’il est possible de réduire le nombre d’instructions exécutées par un algorithme en occupant plus de mémoire. Ce qui signifie que mesurer le nombre d’instruction ou la quantité de mémoire revient (quasiment) au même.

Donc, si l’on souhaite comparer deux algorithmes, on pourrait penser qu’il suffit de comparer le nombre d’instructions qu’ils exécutent.

Sauf que toutes les instructions ne se valent pas.

## Exemples

* Affectation d’une valeur dans une variable

a=42

Il s’agit d’une instruction en pseudo-langage simple qui pourrait être représentée par une unique instruction machine (instruction MOVE). On considère donc que cette instruction est équivalente à une unité de calcul.

* Embranchement conditionnel

b=a==3

si(b){

afficher('Bravo')

}sinon{

afficher('Raté')

}

Un embranchement conditionnel doit au moins stoker la valeur du résultat avant de modifier le pointeur d’exécution. Cette instruction peut donc nécessiter plus d’une unité de calcul pour s’exécuter. Cependant, le nombre d’instructions nécessaires pour tous les embranchements conditionnels est constants.

Donc, si on mesure le nombre d’instructions en pseudo-langage d’un algorithme en utilisant la formule suivante : avec le nombre d’affection le nombre d’embranchements conditionnels…, et que l’on souhaite calculer le nombre d’instructions machines exécutées, il suffit de connaître le nombre d’instructions machines nécessaires à l’exécution de chaque type d’instruction langage :

Avec le nombre d’instructions nécessaires pour l’exécution d’une affectation, le nombre d’instruction nécessaires pour l’exécution d’un embranchement conditionnel…

* Itérations

a=0

tanque (a<10){

afficher(a)

a++

}

Pour les boucles, le nombre d’instruction est dépendant du nombres d’itérations réalisées à l’exécution. Contrairement aux autres instructions en pseudo-langage, les boucles ne sont pas constantes en nombre d’instructions machines. Elles sont proportionnelles au nombre d’itérations .

Notre formule précédente devient donc :

sont des constantes liées au type de langage machine utilisé.

sont des constantes liées à l’algorithme étudié

En revanche, est une variable pour la fonction calculant le nombre d’itérations exécutée.

On va donc compacter l’écriture. On pose , le nombre d’instructions machines pour l’algorithme . Et on écrit : , la fonction de mesurant le nombre d’instructions machines pour l’algorithme

# Mesure de la complexité

## Premières mesures de l’efficacité

### Exemple1

Soient les algorithmes suivants :

fonction A1 (){

entier a=4

entier b=5

afficher(a+b)

}

fonction A2 (){

entier a=4

entier b=5

entier c= a+b

afficher(c)

}

On considère que la fonction d’affichage afficher(x) exécute une seule affectation vers la sortie standard et compte donc comme 1 unité de calcul.

On considère que la déclaration ne compte pour aucune unité de calcul.

Mesurer et

et

Les deux algorithmes répondent au même problème (afficher « 9 ») mais l’algorithme est plus efficace que l’algorithme car

De plus, on constate que ces deux algorithmes sont constants en fonction de .

### Exemple 2

fonction A3 (entier a):bool{

si(a>10){

retourne(vrai)

}sinon{

si(a+4<12){

retourne(vrai)

}sinon{

retourne(faux)

}

}

}

On considère que la fonction retourne(x) compte pour une affectation à la variable de retour de fonction.

On considère qu’un embranchement conditionnel compte pour une unité de calcul par convention.

Mesurer

On observe 3 cas distincts de calcul pour :

* et
* et

On peut réunir les deux derniers cas, car la valeur de est la même

L’algorithme exécutera 2 unités de calcul dans le meilleur des cas (si ) et 3 unités de calcul dans le pire des cas (si ).

Pour déterminer le nombre d’unité de calcul dans le cas général (ou le cas moyen), il faut déterminer la probabilité d’apparition du meilleur et du pire des cas.

Si le meilleur et le pire des cas sont équiprobables (de même probabilité dans notre exemple il y a autant de chance d’avoir que d’avoir ) alors la valeur de pour le cas général est la moyenne du pire et du meilleur des cas :

Sinon, on affectera à chaque cas un coefficient de probabilité pour mesurer le cas moyen :

Avec le nombre d’unité de calcul pour l’algorithme en fonction de dans le cas général

le nombre d’unité de calcul pour l’algorithme en fonction de dans le pire des cas ()

le nombre d’unité de calcul pour l’algorithme en fonction de dans le meilleur des cas (m)

est la probabilité d’apparition du pire des cas

est la probabilité d’apparition du meilleur des cas

Dans la suite de ce cours, et sauf exception, on considérera l’ensemble des cas comme équiprobables.

### Exemple 3

fonction A4 (entier[n] tableau, entier a) :bool{

pour (entier i=0 ;i<n ;i++){

si(a==tableau[i]){

retourne(vrai)

}

}

retourne(faux)

}

On considère que les boucles comptent pour une unité de calcul.

Caractérisation des cas :

* Le pire des cas est celui où la variable a n’appartient pas au tableau
  + On compte 2 unités de calcul (1 pour et 1 si) par itération, et on itère n fois la boucle. Enfin, on retourne(faux), on additionne donc 1
* Le meilleur des cas est celui où la variable a est au début du tableau
  + On compte 3 unités de calcul (1 pour, 1 si, 1 retourne).
* Le cas moyen est la moyenne des cas

### Exercice 1

fonction f1(entier[n] tab){

entier val=tab[0]

entier i=1

tant que(i<= n-1){

si(val<tab[i]){

val=tab[i]

}

i++

}

retourne(val)

}

#### Description de l’algorithme

L’algorithme recherche le maximum d’un tableau

#### Caractérisation des cas

* Le pire des cas est que l’ensemble du tableau est trié par ordre croissant et la valeur la plus grande est à la fin du tableau
* Le meilleur est que la valeur la plus grande du tableau se trouve au premier index.

#### Mesure de l’efficacité du pire des cas

En effet, on compte 3 affectations n-1 fois la boucle, l’embranchement conditionnelle est les deux affections incluses dans le tantque.

#### Mesure de l’efficacité du meilleur des cas

#### Mesure de l’efficacité moyenne

Il n’est pas précisé dans l’énoncé de probabilités d’apparition des cas. On considère donc les pire et meilleur cas comme équiprobable.

On peut en déduire que le cas moyen est d’efficacité :

#### Caractérisation du cas général

Le cas général se présent lorsque ni le pire ni le meilleur cas ne se présente.

Il s’agit du cas où la valeur la plus grande du tableau ne se trouve ni au début ni à la fin du tableau et que celui-ci n’est pas trié par ordre croissant.

Cela signifie que chaque position de la valeur maximale est cas à traiter en fonction de sa probabilité d’apparition.

Comme le maximum ne peut ni être au début ni à la fin d’un tableau de n indices, alors il reste n-2 cas à traiter.

* Premier cas, le maximum est à l’indice 1
* 2ème cas, le maximum est à l’indice 2
* 3ème cas, le maximum est à l’indice 3
* 4ème cas, le maximum est à l’indice 4
  + On a bien 3 affectations en dehors de la boucle
  + On a 3 boucles où on ne sait si les valeurs testées sont inférieures ou supérieures à la valeur (val). Dans ces boucles on a une chance sur deux qu’elle soit inférieure (4 tests conditionnels et affectations) et une chance sur deux qu’elle soit supérieure (3 tests conditionnels et affectations)
  + On a une boucle où la valeur testée courante est le maximum (la 4ème boucle) et donc on est sûr d’utiliser 4 unités de calcul.
  + Pour les boucles restantes, on sait que les valeurs restantes seront systématiquement inférieures à la valeur courante (puisqu’elle est maximum). On a donc 3 tests conditionnels et affectations pour chacune de ces boucles.

On peut constater que, en fonction de la position du maximum, on pourrait écrire que (il faut le démontrer par récurrence)

Au final on aura

### Exercice 2

fonction f2(entier[n] tab, entier i, entier a){

pour entier j allant de i à n-1{

si(tab[j]>a){

afficher(tab[j])

}

}

}

#### Description de l’algorithme

Affiche les valeurs supérieures à une valeur a donnée dans un tableau tab à partir d’un indice i donné.

#### Caractérisations des cas

* Le pire des cas est réalisé si l’ensemble des valeurs du tableaux tab à partir de l’indice i sont toutes supérieures à a.
* Le meilleur des cas est réalisé si l’ensemble des valeurs du tableaux tab à partir de l’indice i sont toutes inférieures ou égales à a.
* Le cas moyen est réalisé quand la moitié des valeurs du tableau tab à partir de l’indice i sont toutes supérieures à a ou inférieures ou égales à a.

#### Mesure de l’efficacité pour le meilleur des cas

On constate que l’algorithme n’a aucune opération en dehors de la boucle pour.

La boucle est exécutée fois car si on exécute fois la boucle et si on effectue fois la boucle.

On note le nombre d’unités de calcul réalisées dans chaque boucle.

On peut donc écrire :

Or dans le meilleur des cas car on compte le test conditionnel dans le pour et celui du si

On en déduit que

#### Mesure de l’efficacité pour le pire des cas

On constate que l’algorithme n’a aucune opération en dehors de la boucle pour.

La boucle est exécutée fois car si on exécute fois la boucle et si on effectue fois la boucle.

On note le nombre d’unités de calcul réalisées dans chaque boucle. On peut donc écrire :

Or dans le meilleur des cas car on compte l’affichage qui se fera à chaque fois ainsi que le test conditionnel dans le pour et celui du si

On en déduit que

#### Mesure de l’efficacité pour le cas moyen

On constate que l’algorithme n’a aucune opération en dehors de la boucle pour.

La boucle est exécutée fois car si on exécute fois la boucle et si on effectue fois la boucle.

On note le nombre d’unités de calcul réalisées dans chaque boucle. On peut donc écrire :

Or dans le meilleur des cas car une fois sur deux, la valeur du tableau à l’indice j est inférieure ou égale à a et une fois sur deux elle est supérieure à a.

On en déduit que

## Mesure de la complexité

On mesure la complexité en fonction de la taille des données passées en paramètre de l’algorithme (taille du ou des tableaux passés en paramètre).

On considère que cette mesure est équivalente au dénombrement des unités de calcul réalisées l’algorithme.

On distingue les 3 cas de mesures :

* Optimum : le meilleur des cas qui réalise le moins d’unités de calcul
* Maximum : le pire des cas qui réalise le plus d’unités de calcul
* Moyen : qui sera calculé en fonction des cas optimum et maximum et de leur probabilité d’apparition.

### Démarche générique de la mesure de la complexité :

1. On détermine le problème auquel répond l’algorithme
2. On décrit textuellement les cas optimum et maximum
3. On calcul pour chaque cas le nombre d’unité de calcul nécessaire en fonction de la taille des données d’entrée
4. On calcule le cas moyen en fonction de la probabilité d’apparition des cas.

On note la fonction de complexité d’un algorithme.

### Complexité algorithmique et problèmes mathématiques

Un algorithme par définition répond à une problématique mathématique.

Les programmes informatiques sont donc des implémentations de solutions à des problèmes mathématiques, c’est pourquoi, pour répondre à des problématiques métiers, on passe souvent par une formalisation mathématique.

De fait, on considère que la complexité d’un algorithme qui résout un problème est équivalente à la complexité du problème résolu si l’algorithme est celui qui présente la complexité la plus faible des algorithmes répondant au problème.

Soit l’ensemble des fonctions de complexité des algorithmes répondant à un problème. Alors , la complexité du problème peut s’écrire :

L’avantage de ce principe est que la découverte d’une solution moins complexe permettant de résoudre un problème réduit mécaniquement la complexité de celui-ci.

## Méthodes de calcul de la complexité

### Appel séquentiel

fonction f(){

g()

h()

}

### Appel conditionnel

fonction f(){

si(…){

g()

}sinon{

h()

}

}

Dans le meilleur des cas :

Dans le pire des cas :

### Appel itératif

fonction f(…,entier n){

pour(entier i=n0 ;i<n ;i+=k){

g()

}

}

/!\ATTENTION /!\

Si la fonction g prend en paramètre l’incrémentateur i, alors la formule de complexité change

fonction f(…,entier n){

pour(entier i=n0 ;i<n ;i+=k){

g(…,i)

}

}

Soit la complexité de la boucle .

Soit la complexité de la fonction g(…,i) en fonction de la boucle courante .

La complexité de la première boucle, quand est :

La complexité de la deuxième boucle, quand est :

La complexité de l'avant-dernière boucle, quand est :

La complexité de la dernière boucle, quand est :

On obtient donc pour la formule suivante :

On peut réunir l'ensemble des termes constant à gauche et le reste des termes à droite de la formule. On obtient donc :

Les parenthèses contiennent autant de termes que de boucles parcourues soit boucles.

Soit le signe sigma permettant de définir des sommes d'élément similaire.

Exemple :

Autre exemple avec un pas :

On en déduit l'écriture suivante pour la fonction de complexité de f():

# Exercices de mesure de la complexité

Mesurer la complexité dans le meilleur cas, pire des cas, et en moyenne pour les fonctions suivantes :

fonction f1(entier[n] tab){

entier val=tab[0]

entier i=1

tant que(i<= n-1){

si(val<tab[i]){

val=tab[i]

}

i++

}

retourne(val)

}

fonction g1(entier val, entier valTab){

si(val< valTab){

val= valTab

}

}

fonction f1Reecrite(entier[n] tab){

entier val=tab[0]

entier i=1

tant que(i<= n-1){

g1(val,tab[i])

i++

}

retourne(val)

}

#### Description de l’algorithme

L’algorithme recherche le maximum d’un tableau

#### Caractérisation des cas

* Le pire des cas est que l’ensemble du tableau est trié par ordre croissant et la valeur la plus grande est à la fin du tableau
* Le meilleur est que la valeur la plus grande du tableau se trouve au premier index.

#### Mesure de la complexité

La complexité de la fonction f1() est la somme de la complexité des deux affectations () avec la complexité de l'appel itératif de la fonction g1() et d'une affection sommée ( d'après la formule Appel itératif) avec la complexité de la fonction retourne.

Il nous reste à déterminer la complexité des fonctions g1() et retourne.

Or la fonction g1() contient un appel conditionnel. Elle va donc avoir une complexité différente en fonction de son pire ou meilleur cas.

#### Mesure de la complexité de g1() dans le meilleur cas

D'après la formule Appel conditionnel, on peut écrire que la complexité de g1() est :

#### Mesure de la complexité de g1() dans le pire cas

D'après la formule Appel conditionnel, on peut écrire que la complexité de g1() est :

#### Mesure de la complexité g1() dans le cas moyen

On étend la formule Appel conditionnel, en déterminant la valeur du cas moyen comme la moyenne des pire et meilleur cas :

#### Mesure de la complexité de f1() dans le meilleur cas

On en déduit que dans le meilleur des cas est :

Comme vu précédemment (dans l'Exemple 2 du cours), et sans précision autre, on considère que

On en déduit la valeur de dans le meilleur des cas :

#### Mesure de la complexité de f1() dans le pire cas

On en déduit que dans le pire des cas est :

On en déduit la valeur de dans le pire des cas :

#### Mesure de la complexité de f1() dans le cas moyen

On en déduit que dans le cas moyen est :

On en déduit la valeur de dans le cas moyen :

fonction f2(entier[n] tab, entier i, entier a){

pour entier j allant de i à n-1{

g2(a,tab[j])

}

}

fonction g2(entier a, entier val){

si(val>a){

afficher(val)

}

}

#### Description de l’algorithme

Affiche les valeurs supérieures à une valeur a donnée dans un tableau tab à partir d’un indice i donné.

#### Caractérisations des cas

* Le pire des cas est réalisé si l’ensemble des valeurs du tableaux tab à partir de l’indice i sont toutes supérieures à a.
* Le meilleur des cas est réalisé si l’ensemble des valeurs du tableaux tab à partir de l’indice i sont toutes inférieures ou égales à a.
* Le cas moyen est réalisé quand la moitié des valeurs du tableau tab à partir de l’indice i sont toutes supérieures à a ou inférieures ou égales à a.

#### Mesure de la complexité

La complexité de la fonction f2() est la complexité de l'appel itératif de la fonction g2() ( d'après la formule Appel itératif).

Il nous reste à déterminer la complexité de fonction g2().

Or la fonction g2() contient un appel conditionnel. Elle va donc avoir une complexité différente en fonction de son pire ou meilleur cas.

#### Mesure de la complexité de g2() dans le meilleur cas

D'après la formule Appel conditionnel, on peut écrire que la complexité de g2() est :

Or on a fixé plus tôt que la fonction afficher réalisait 1 unité de calcul (dans l'Exemple1 du cours).

On obtient donc :

#### Mesure de la complexité de g2() dans le pire cas

D'après la formule Appel conditionnel, on peut écrire que la complexité de g2() est :

#### Mesure de la complexité g2() dans le cas moyen

On étend la formule Appel conditionnel, en déterminant la valeur du cas moyen comme la moyenne des pire et meilleur cas :

#### Mesure de la complexité de f2() dans le meilleur cas

On en déduit que dans le meilleur des cas est :

#### Mesure de la complexité de f2() dans le pire cas

On en déduit que dans le pire des cas est :

#### Mesure de la complexité de f2() dans le cas moyen

On en déduit que dans le cas moyen est :

fonction f3(entier[n] tab, entier j) entier{

entier val=j

entier i=j+1

tant que(i<= n-1){

g3(tab[val],tab[i],val,i)

i++

}

retourne(val)

}

fonction g3(entier valTab1,entier valTab2,entier val,entier i){

si(valTab1<valTab2){

val=i

}

}

#### Description de l’algorithme

L'algorithme retourne l'indice de la valeur maximum d'un tableau passé un indice j donné.

#### Caractérisations des cas

* Le meilleur des cas correspond à celui où les appels à la fonction g3() exécutent le moins d'unités de calcul, c’est-à-dire, si la valeur maximale est la valeur située à l'indice j.
* Le pire des cas est celui où les appels à la fonction g3() exécutent le plus d'unités de calcul, c’est-à-dire, si la valeur maximale est située à la fin du tableau et les valeurs sont triées par ordre croissant à partir de l'indice j.
* Le cas moyen est celui où le tableau n'est pas trié à partir de l'indice j et la valeur maximale est à un indice quelconque.

#### Mesure de la complexité

La complexité de la fonction f3() est somme la complexité de l'appel itératif de la fonction g3() ajouté à une affectation (i++), avec la complexité de la fonction retourne.

Or nous sommes dans le cas où la fonction g3() prend en paramètre l'indice i.

On obtient donc pour la boucle la fonction de complexité suivante (d'après celle de l'Appel itératif) :

Et donc

Il nous reste à déterminer la complexité des fonctions g3() et retourne.

Or la fonction g3() contient un appel conditionnel. Elle va donc avoir une complexité différente en fonction de son pire ou meilleur cas.

#### Mesure de la complexité de g3() dans le meilleur cas

D'après la formule Appel conditionnel, on peut écrire que la complexité de g3() est :

#### Mesure de la complexité de g3() dans le pire cas

D'après la formule Appel conditionnel, on peut écrire que la complexité de g3() est :

#### Mesure de la complexité g3() dans le cas moyen

On étend la formule Appel conditionnel, en déterminant la valeur du cas moyen comme la moyenne des pire et meilleur cas :

On constate que quelle que soit la valeur de , et quels que soient les cas, la fonction de complexité pour la fonction g3() est indépendante de .

#### Mesure de la complexité de f3() dans le meilleur cas

Dans le meilleur des cas, la valeur maximale du tableau tab est à l'indice j.

On exécute donc le pire des cas de g3() à la première itération de la boucle et le meilleur des cas de g3() sur les boucles suivantes.

On obtient donc pour la formule :

Comme vu précédemment (dans l'Exemple 2 du cours), et sans précision autre, on considère que

On a donc

#### Mesure de la complexité de f3() dans le pire cas

Dans le pire des cas, la valeur maximale du tableau tab est au dernier indice et le tableau est trié.

On exécute donc le pire des cas de g3() sur toutes les itérations.

On obtient donc pour la formule :

On a donc

#### Mesure de la complexité de f3() dans le cas moyen

Dans le pire des cas, le tableau tab n'est pas trié et l'indice de la valeur maximale est quelconque.

Soit l'indice de la valeur maximale du tableau avec

On exécute g3() dans le cas moyen pour les itérations allant de à et dans le meilleur des cas des itérations allant de à .

On obtient donc pour la formule :

On a donc

fonction f4(entier[n] tab){

entier j,k

pour i allant de 0 à n-1{

j=f3(tab,i)

k=tab[i]

tab[i]=tab[j]

tab[j]=k

}

}

fonction f5(entier[n] tab){

pour i allant de 0 à n-1{

temp(i,tab)

}

}

fonction temp(entier i, entier[n] tab){

pour j allant de i à n-1{

si(tab[j]<=tab[i]){

k=tab[i]

tab[i]=tab[j]

tab[j]=k

}

}

}

fonction union(entier[n] tab1,entier[m] tab2):entier[]{

entier[] ret

entier k=0

pour i dans tab1{

pour j dans tab2{

si(i!=j){

ret[k]=j

k++

}

ret[k]=i

k++

}

}

retourne ret

}

# Classes de complexité

Les algorithmes peuvent être classés par catégorie que l’on appelle « classe de complexité ».

Pour se faire nous allons utiliser un outil mathématique appelé « majoration (minoration) asymptotique »

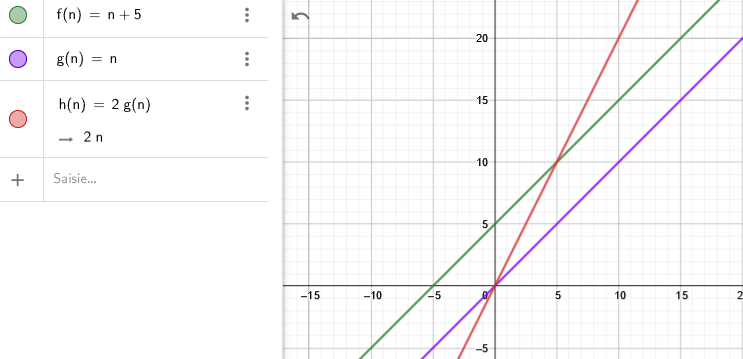
Majorer en asymptote revient à chercher une fonction pour laquelle, à partir d’un n assez grand, cette dernière est toujours supérieure à toutes les fonctions de sa classe.

## Majoration asymptotique

Soit une fonction positive ou nulle

Soit l’ensemble des fonctions majorées par à une constante près et pour un assez grand (asymptotique).

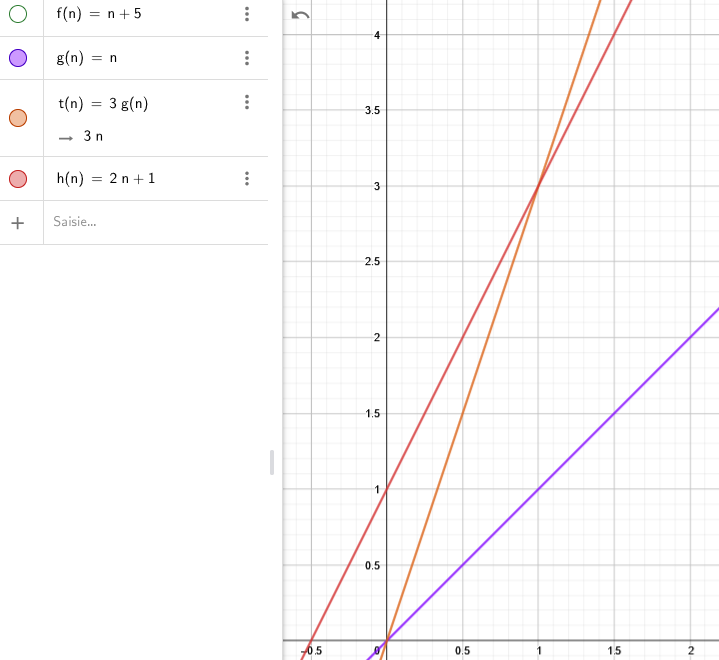
### Exemple graphique sur Geogebra



Ici on voit que la fonction fait parti de avec car il existe et telles que pour tout .

Qu’en est-il de la fonction ?

Si on choisit et alors on obtient bien



### A quoi ça sert ?

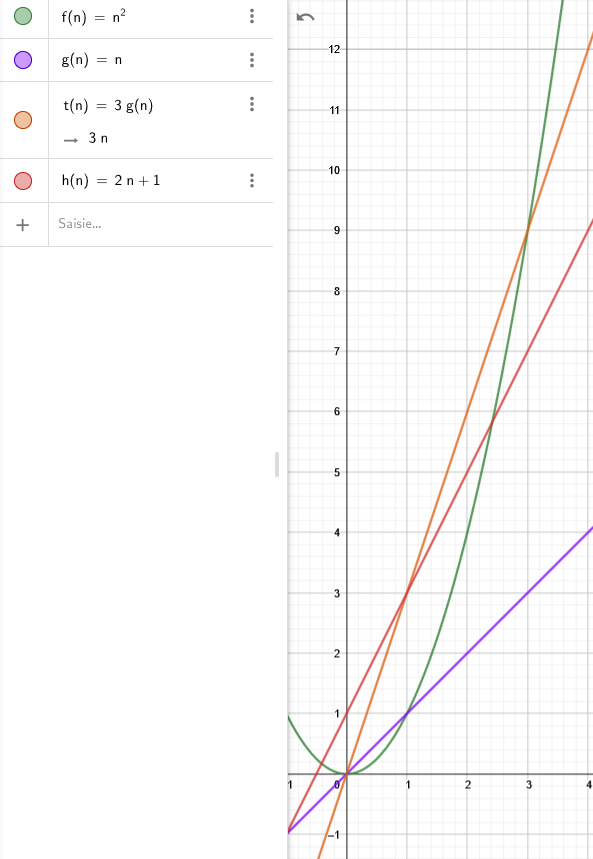
On va donc pouvoir classer les fonctions par rapport aux fonctions qui les majore.

Ce qui signifie, si l’on parle de fonction de complexité, qu’une fonction de complexité appartenant à ne pourra jamais avoir une complexité plus grande que à une constante près.

On va aussi pouvoir déterminer quelles fonctions n’appartiennent pas à .

Ces fonctions n’appartiendront pas à la classe des fonctions de complexité maximum en

### Exemple graphique avec Geogebra



Ici, si alors il est impossible de trouver un tel que

Donc a une complexité qui sera toujours supérieure à celle de l’ensemble des fonctions appartenant à

On peut donc éliminer la fonction dont la complexité est représentée par comme une fonction améliorant la réponse à un problème de complexité

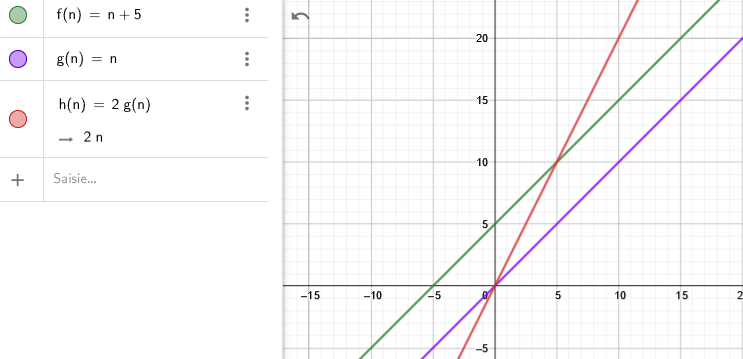
On peut même ajouter que l’ensemble des problèmes de complexité ont des solutions de qui sont décrit par des fonction de complexité appartenant à

## Minoration asymptotique

Soit une fonction positive ou nulle

Soit l’ensemble des fonctions minorée par à une constante près et pour un assez grand (asymptotique).

### Exemple graphique sur Geogebra



Ici la fonction est toujours inférieure à la fonction , c’est donc qu’il existe et tels que

Même calcul pour

Les deux fonctions appartiennent donc à

C’est-à-dire que, si ces fonctions sont des fonctions de complexité, les algorithmes qu’elles représentent ne pourront pas avoir une complexité inférieure à à une constante près

Ce qui signifie que les algorithmes dont les fonctions de complexité appartiennent à et à alors leur complexité ne pourra jamais être supérieure ou inférieure (on dit qu’elles sont bornées) à

Exemple si

Alors les problèmes de complexité appartenant à et à sont dit de complexité

De plus, tous les problèmes dont la complexité n’appartient pas à et à sont dit de complexité supérieure ou inférieure à (en fonction de leur classe de complexité)

On peut donc, si l’on connaît la complexité des problèmes à résoudre par un algorithme, on peut savoir si notre solution est de la même classe de complexité et donc répond de manière satisfaisante.

En revanche si découvre que la complexité de la solution est supérieure à celle du problème, c’est que l’algorithme peut être optimisé.

Et surtout si on découvre une solution dont la complexité est inférieure à celle problème, alors le problème devient de la complexité de la solution trouvé.

Rappel : , la complexité du problème peut s’écrire :

# Signification de l’appartenance à et à

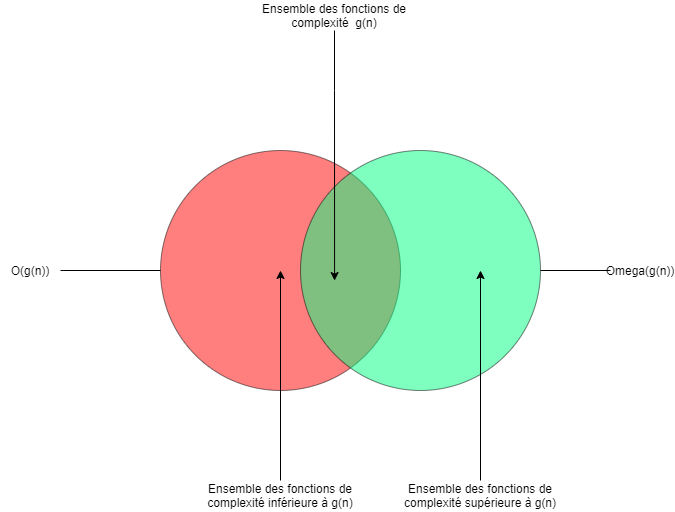


Figure : représentation des ensembles de complexité

On peut déduire du schéma que

* Si une solution à un problème de complexité dont la fonction complexité appartient à mais pas à
  + La solution est de complexité inférieure au problème auquel elle répond signifiant que
    - * Le problème de complexité inférieure à (rare et dur à démontrer)
    - Ou
      * Cette solution est partielle (ne prend pas tous les cas)
    - Ou
      * Cette solution est incorrecte
* Si une solution à un problème de complexité dont la fonction complexité appartient à mais pas à
  + La solution est de complexité supérieure au problème auquel elle répond signifiant qu’elle peut être optimisée et on est sûr d’en être capable
* Si une solution à un problème de complexité dont la fonction complexité appartient dans le pire des cas à et dans le meilleur des cas à
  + Alors la solution en moyenne est de complexité
  + On dit qu’elle appartient à l’ensemble des fonctions positive ou nulle bornées par à deux constantes près et en asymptote.
  + Cet ensemble s’écrit :

On peut aussi écrire que

Donc pour déterminer la classe d’appartenance d’une fonction de complexité il faut et il suffit de déterminer son appartenance à et

# Les grandes familles de classe complexité

|  |  |  |
| --- | --- | --- |
| **Classe de complexité** | **Complexité** | **Exemple** |
| avec une constante | constante | Interversion de valeurs |
|  | logarithmique | Recherche dichotomique |
|  | linéaire | Parcours de tableau |
|  | Quasi-quadratique |  |
|  | quadratique | Parcours d’une matrice |
| avec une constante | polynomiale | Imbrication de boucles |
| avec une constante | Exponentielle | Factorisation en nombres premiers |