

Algorithmique et Analyse d'Algorithmes

L3 Info

Cours 1 : notion de coût d'un algorithme

Benjamin Wack



2025 – 2026

Plan

Présentation du cours

Problèmes et algorithmes

Coût d'un algorithme

Complexité

- Méthodologie

- Ordres de grandeur

Un problème, plusieurs algorithmes

Objectifs du cours

Savoir **proposer** une solution **algorithmique** à un **problème** posé, savoir **implémenter** la solution et savoir **analyser** celle-ci.

Objectifs détaillés

- ▶ Savoir **reconnaître** et mettre en œuvre des **schémas génériques** d'algorithmes (parcours d'arbre, algo glouton...),
- ▶ Savoir **construire** une solution selon une démarche allant du plus simple (algorithme naïf) au plus efficace (diviser pour régner, etc.)
- ▶ Savoir **démontrer** la correction des algorithmes
- ▶ Savoir comment **évaluer** la complexité d'une solution algorithmique :
 - analyser la complexité au pire, en moyenne avec des **hypothèses probabilistes**,
 - analyser la complexité en utilisant des mesures sur des simulations ou des **jeux de test**.

Structure du cours

Cours

Une partie synthétique sur les **concepts** et schémas algorithmiques
Un **algorithme classique** afin de se constituer une culture de référence

TD1

Exercices permettent de **renforcer la compréhension** des concepts.

TD2

Implémentation des structures et **préparation** aux activités pratiques

APNEES : Activités Personnelles Non Encadrées Évaluées

Validation des concepts par la pratique et évaluation de la **compréhension**

+ le travail personnel ! (exercices, petits programmes...)

Adresse Mail enseignant : `Prenom.Nom@univ-grenoble-alpes.fr`

Ressources

Bibliographie

- ▶ *Algorithmique*, T. Cormen, R. Rivest & C. Leiserson, Dunod
- ▶ *Algorithmes*, R. Sedgewick, Pearson Education

Parcours Moodle de l'UE

- ▶ Planning, documents, annales
- ▶ Sujets et rendus d'Apnées
- ▶ Ponctuellement : activités auto-évaluées

<https://moodle.caseine.org/course/view.php?id=1051>

(dans vos cours sur Caséine, par le Moodle de la L3, par ma page Web...)

Évaluations

Note finale = 65 % Examen + 35 % CC

Les contrôles continus

- ▶ 2 quicks (début octobre et début novembre) : 10 % chacun
- ▶ 3 comptes-rendus d'APNEEs : 15 % au total

Examen terminal

- ▶ 2h30 pas de calculatrice, 1 feuille A4 recto verso
- ▶ Session 2 en juin...

Programme (indicatif) du cours

► Complexité des algorithmes

1. Coût d'un algorithme (itérations, ordres de grandeur) *La star*
2. Récursivité, analyse en moyenne *Quicksort*

► Types abstraits élémentaires et implantation

3. Structures séquentielles *Algorithme de parenthésage*
4. Structures arborescentes *Partition Binaire de l'Espace*

► Preuves d'algorithmes

5. Invariant, correction, terminaison *Drapeau hollandais*
6. Logique de Hoare *Dichotomie*

► Arbres

7. Arbres binaires de recherche *Arbres B*
8. Arbres ordonnés, structure de tas
9. Arbres et codage *Algorithme de Huffman*

► Graphes et algorithmes

10. Algorithmes gloutons *Coloration de graphes*
11. Algorithmique de graphes *Prim et Kruskal*

Notion de **problème**

données

- ▶ Effectuer une recherche dans un inventaire *stock*
- ▶ Résoudre une équation *coefficients*
- ▶ Trouver le plus court chemin sur un plan de ville *plan*
- ▶ Corriger des fautes d'orthographe *texte + dictionnaire*
- ▶ Multiplier des matrices *coefficients*
- ▶ ...

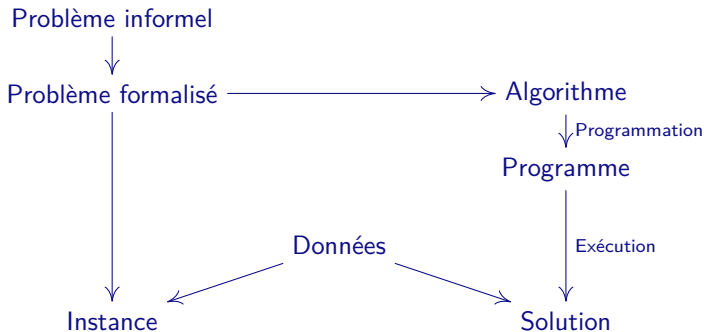
Dans chacun de ces exemples on s'intéresse en fait à une **classe** de problèmes similaires, dont chaque **instance** est définie par des **données**.

Il faut également préciser le **résultat** attendu.

(par exemple pour la correction orthographique : signaler les fautes ? les corriger ?)

Qu'est-ce qu'un algorithme ?

Procédure de résolution de **n'importe quelle instance** d'un problème suffisamment élémentaire pour être exécutée de façon automatique.



Exemple : correcteur orthographique

Un algorithme **traite** donc des données pour produire un résultat ; mais avec quelles **primitives** et quelles **ressources** ?

S'il fallait tenir compte :

- ▶ du matériel
- ▶ du système d'exploitation
- ▶ du langage de programmation

on passerait son temps à réinventer les mêmes algorithmes.

Algorithme versus programme

Un algorithme décrit une méthode générale de résolution d'un problème :

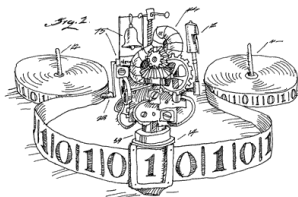
- ▶ que l'on pourra traduire dans **n'importe quel** langage de programmation
- ▶ que l'on pourra **analyser** pour en dégager les caractéristiques

Contre-exemples : tri spaghetti

Nécessité d'un **modèle**



Alan M. Turing
1912-1954



Machine « de papier »
1936

La machine est dotée :

- ▶ d'un ruban infini (\simeq mémoire)
- ▶ d'une tête de lecture/écriture
- ▶ d'un automate (\simeq processeur)

Un modèle donc tourné vers les opérations de lecture/écriture
(\simeq affectation, calculs)

Notre modèle de machine

▶ Mémoire

- infinie (mais un calcul donné utilise un espace fini),
- types élémentaires (finis) `int`, `float`, ...
- puis on ajoutera des structures de données

▶ Opérations

- nombre fini d'opérations (arithmétiques, booléennes...);
- chaque opération a un nombre fini de paramètres
(nombre fini de variables lues et écrites)

▶ Processeur

- processeur unique;
- effectue les opérations en temps constant ($1 \text{ top} = 1 \text{ opération}$)

Les questions abordées en Algo5

Est-ce que je peux construire un programme qui résout mon problème (sur une machine conforme à mon modèle) ?

Spécification du problème et construction d'un algorithme

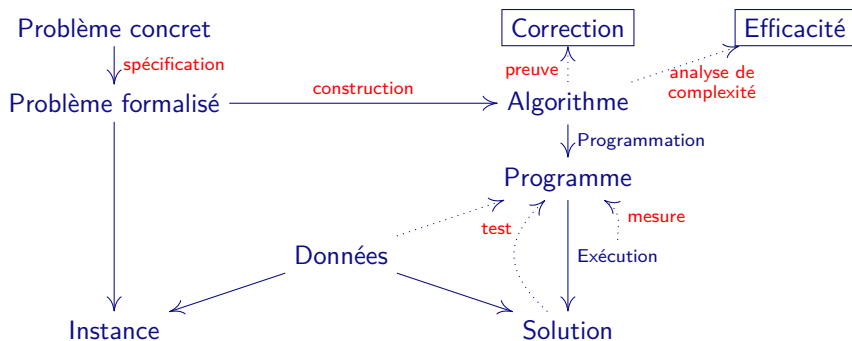
Est-ce que l'exécution de mon programme sur la machine donne bien le résultat souhaité ?

Vérification de propriétés qualitatives par test et preuve

Est-ce que mon programme me fournit le résultat en un temps acceptable ?

Validation de propriétés quantitatives par mesure et analyse

Les questions abordées en Algo5



Exemple : correction orthographique

Effacité d'un algorithme

- ▶ Est-ce que mon programme consomme une quantité **acceptable** de ressources pour fournir un résultat ?
- ▶ Étant donnés deux programmes résolvant le même problème, lequel est le "**meilleur**" ?

Notion(s) de coût(s)

- ▶ coût en temps
= nombre d'opérations permettant d'obtenir le résultat à partir des données
- ▶ coût en mémoire
= nombre maximal de variables utilisées simultanément durant le calcul

Exemple (1)

Maximum de 3 éléments

MAX(a, b, c)

Données : Trois éléments de même type (comparable) a , b et c

Résultat : Le plus grand des trois

$$m \geq a \text{ et } m \geq b \text{ et } m \geq c \quad \text{et} \quad m = a, b \text{ ou } c$$

si $a > b$

└ $m := a$

sinon

└ $m := b$

si $c > m$

└ $m := c$

Renvoyer m

Modèle de coût ?

Ne comptons que les
comparaisons entre éléments

Coût de l'algorithme

Coût temps ? 2 comparaisons + 1 ou 2 affectations

Coût en espace mémoire ? 1 variable

Coût constant, indépendant des données

Exemple (2)

Comptage d'un élément

OCCURRENCES(x, T)

Données : Un élément x et un tableau T de taille n

Résultat : Le nombre d'occurrences de x dans T

$k := 0$

pour $i := 0$ à $n - 1$

┌ **si** $T[i] = x$
└ $k := k + 1$

Renvoyer k

- ▶ Coût en temps : n comparaisons entre éléments
(+ les opérations sur les indices)
- ▶ Coût variable, dépend de la **taille** des données
mais toujours n comparaisons pour un tableau de taille n

Exemple (3)

Vérification du tri

VERIF_TRI(T)

Données : un tableau T de n
éléments

Résultat : T est-il trié ?

$i := 0$

tant que $i < n - 1$ et $T[i] \leq T[i + 1]$
 \perp $i := i + 1$

Renvoyer ($i = n - 1$)

Coût de l'algorithme

(toujours en comparaisons
entre éléments de T)

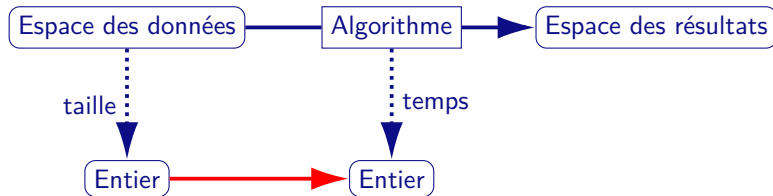
- ▶ pour $T = [1, 2, \dots, 10]$? 9
- ▶ pour $T = [1, 2, \dots, n]$? $n - 1$
- ▶ pour $T = [2, 1, \dots]$? 1

Dépend de la **taille** des données
Dépend aussi de la **valeur** des
données

Le coût calculé doit donner une information *générale*

- ▶ Il s'exprime toujours en fonction de la *taille* des données
- ▶ Si trop de variation on cherche une *borne supérieure* (pire cas)

Généralisation



Caractériser l'efficacité générale de l'algorithme = trouver la **relation** entre la **taille des données** et le **temps d'exécution de l'algorithme** sur un (modèle de) machine donnée

Remarque : attention aux entiers, le temps de calcul de x^n dépend de n qui n'est pas la taille de la donnée.

Coût d'un algorithme

Coût (au pire)

Le coût d'un algorithme \mathcal{A} est **fonction** de la **taille** des données :

$$C_{\mathcal{A}}(n) = \max(\text{coût}(d)) \text{ pour toutes les données } d \text{ de taille } n$$

- ▶ Suppose d'avoir fixé la notion de taille
- ▶ Maximum = **garantie** quelles que soient les conditions d'utilisation
- ▶ Concrètement : donner une **borne supérieure** du coût
+ exhiber un **cas défavorable**

Coût au mieux

$$C_{\mathcal{A}}^{\min}(n) = \min(\text{coût}(d)) \text{ pour toutes les données } d \text{ de taille } n$$

- ▶ Correspond au cas le plus favorable

Laquelle de ces deux informations est la plus utile ?

Coûts des structures de base

Principe du calcul de coût

Le calcul du coût d'un algorithme s'obtient en **composant** les coûts des différentes opérations composant l'algorithme.

- ▶ décomposition et reconstruction

Instructions en séquence

Si un algorithme s'écrit sous la forme

Faire Truc

Faire Machin

Son coût est la somme des coûts de Truc et de Machin

Coûts des structures de base (2)

Instructions conditionnelles (IF, SWITCH,...)

Si un algorithme s'écrit sous la forme

si *condition*

└ Faire Truc

sinon

└ Faire Machin

Coût de *l'une* des branches + le coût d'évaluation de la condition

Majoration du coût

Pour le coût au pire on prend donc le maximum des coûts de chaque branche.

$$\text{Coût}(\text{si Condition alors } A \text{ sinon } B) \leq \text{Coût}(\text{évaluation}(\text{Condition})) + \max\{\text{Coût}(A), \text{Coût}(B)\}$$

Coûts des structures de base (3)

Coût d'une boucle (**FOR, WHILE, REPEAT,...**)

pour $i := 1$ à n

└ Faire Truc

tant que *condition*

└ Faire Truc

Somme des coûts de chaque itération

Deux difficultés

- ▶ Chaque itération ne coûte pas forcément la même chose
- ▶ Nombre d'itérations pas toujours connu à l'avance

Coût identique à chaque itération

$n \times \text{coût}(\text{Truc})$

Coût variable

coût(Truc pour $i = 1$)
 + coût(Truc pour $i = 2$)
 + ...
 + coût(Truc pour $i = n$)

si n est connu ou majorable

Exemple

Recherche d'un mot $M[0..k-1]$ dans un texte $T[0..n-1]$

a	a	c	a	b	a	c	a	b	a	a	b	a	a	a
			a	b	a	a								

pour $i := 0$ à $n - k$ (il y a $n - k + 1$ itérations)
 | $j := 0$
 | **tant que** $j < k$ et $T[i + j] = M[j]$ } $1 \leq \text{coût} \leq k$
 | | $j := j + 1$
 | **si** $j = k$ **alors** } négligeable
 | | **Afficher** i
 | $i := i + 1$

Coût de l'algorithme

- ▶ Entre $n - k + 1$ et $k \times (n - k + 1)$ Niveau de détail judicieux ?
- ▶ Pire cas pour des mots du type $M = bbba$ et $T = bbbbbbbbbbbb$

Ordres de grandeur

- ▶ La complexité est une prédiction du temps d'exécution du programme codant l'algorithme.
- ▶ Mais ce temps dépend de l'architecture de la machine, donc c'est une abstraction (approximation)
- ▶ On s'intéresse au passage à l'échelle des algorithmes plus qu'à une mesure précise du temps d'exécution

Pause courbes (1)

Ce qui est important c'est :

1. l'ordre de grandeur ;
2. de pouvoir comparer les algorithmes

Complexité d'un algorithme

On appelle **complexité** d'un algorithme une fonction de référence (logarithme, polynome, exponentielle...) comparable à son coût.

Exemples

 $n = 10^6$

- ▶ moyenne des éléments d'un tableau de taille n
⇒ complexité en n opérations

1/1000^e s

- ▶ tri par insertion des éléments d'un tableau de taille n
⇒ complexité en n^2 opérations

1/4 h

- ▶ énumération des vecteurs de bits de taille n
⇒ complexité en 2^n opérations

 $10^{300\ 000}$ années**Objectif : définir des ordres de grandeur comparables**

Pour se donner une représentation concrète : sur un PC récent, environ 1 milliard d'opérations / seconde

Borne supérieure asymptotique

Notation \mathcal{O}

Pour une fonction donnée g on note $\mathcal{O}(g)$ l'ensemble de fonctions :

$$\mathcal{O}(g) = \{f \text{ telles que } \exists c \geq 0, \exists n_0 \geq 0, \forall n \geq n_0, \quad f(n) \leq c \cdot g(n)\}$$

Par abus de notation on écrit $f = \mathcal{O}(g)$.

On dit que g est une borne supérieure asymptotique pour f .

Vite dit : *f est dépassée par g à partir d'une certaine taille de données*

Exemples

$$3 \times n + 1 = \mathcal{O}(n)$$

$$n + 5\sqrt{n} = \mathcal{O}(n)$$

$$42n + 5 = \mathcal{O}(3n + 42)$$

et réciproquement !

Échelles de comparaison (à connaître)

Échelle logarithmique

▶ $\log(n) = \mathcal{O}(n)$

Échelle polynomiale

▶ Si $p \leq q$ alors $n^p = \mathcal{O}(n^q)$

Exemples

$$n = \mathcal{O}(n^2)$$

$$n^2 = \mathcal{O}(n^3)$$

Échelle exponentielle

▶ Si $0 < a \leq b$ alors $a^n = \mathcal{O}(b^n)$

▶ Tout polynôme est dominé par toute exponentielle

Exemples

$$2^n = \mathcal{O}(3^n)$$

$$n^3 = \mathcal{O}(2^n)$$

Deux propriétés utiles (à connaître)

Coefficient constant

► $k \times f = \mathcal{O}(f)$

Exemple

$$5n^2 = \mathcal{O}(n^2)$$

Addition

► Si $f = \mathcal{O}(g)$ alors $g + f = \mathcal{O}(g)$

Exemple

$$n + \log n = \mathcal{O}(n)$$

$$n^3 + 10n = \mathcal{O}(n^3)$$

Problème : trouver la star

Soit un groupe de n personnes numérotées de 1 à n .

Une personne i connaît j ou bien elle ne la connaît pas.

Une **star** est une personne :

- ▶ que tout le monde connaît
- ▶ mais qui ne connaît personne

Spécifions le problème :

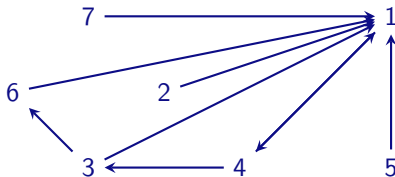
Résultat : Si on renvoie un indice, c'est une star.
Sinon, il n'y a aucune star.

} *Postcondition*

Données : L'entier n , « qui connaît qui »

Modèle de coût :

On comptera le nombre de questions « *est-ce que i connaît j ?* »



Algorithme naïf

naïf = lent mais assez simple pour se persuader qu'il est correct

```
STAR_NAIF( $n$ )
```

```
  pour  $i := 1$  à  $n$ 
```

```
     $i_{star} := true$ 
```

```
    pour  $j := 1$  à  $n$ 
```

```
      si  $j \neq i$  et  $j$  ne connaît pas  $i$ 
```

```
         $i_{star} := false$ 
```

```
    pour  $j := 1$  à  $n$ 
```

```
      si  $j \neq i$  et  $i$  connaît  $j$ 
```

```
         $i_{star} := false$ 
```

```
  si  $i_{star}$ 
```

```
    Renvoyer  $i$ 
```

```
Renvoyer « pas de star »
```

} (*i est-elle connue de tous ?*)

} (*i connaît-elle quelqu'un ?*)

Complexité en $\mathcal{O}(n^2)$

Faire mieux

Peut-on accélérer cet algorithme ?

Factoriser les 2 boucles **for** j en une seule

Non : on économise quelques comparaisons (entre indices) mais on ne pose pas moins de questions.

Remplacer les boucles **for** par des boucles **while**

Non plus : on gagne quelques questions en général, mais dans le pire des cas on reste en $\mathcal{O}(n^2)$.

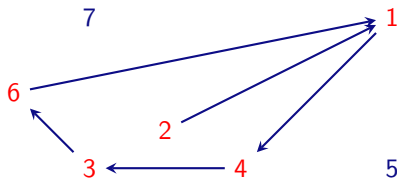
Mieux exploiter les réponses aux questions

Oui :

- ▶ Éviter de poser deux fois la même question
- ▶ Si i connaît j , alors i n'est pas une star

Une tentative

Idee : suivre les liens « ... connaît ... ».



Mais après... ???

La priorité : résoudre le problème

Mieux vaut un algorithme :

- ▶ lent mais correct
- ▶ que « optimisé » mais qui fait n'importe quoi...

Reprenons calmement

- ▶ Si i connaît j , alors i n'est pas une star
- ▶ Si i ne connaît pas j , alors j n'est pas une star

On devrait donc pouvoir éliminer une candidate à *chaque question*.

```
STAR_EFFICACE( $n$ )
```

```
 $i := 1$  // On espère qu'elle ne connaît personne
```

```
 $j := n$  // On espère que tout le monde la connaît
```

```
tant que  $j > i$ 
```

```
  si  $i$  connaît  $j$  //  $i$  n'est pas assez snob
  |  $i := i + 1$ 
```

```
  sinon //  $j$  n'est pas assez connue
  |  $j := j - 1$ 
```

```
// Puis vérifier (comme avant)
```

```
// - si  $i$  est connue de tous
```

```
// - si  $i$  ne connaît personne
```

Complexité en $\mathcal{O}(n)$

Conclusion

- ▶ Un même **problème** peut être résolu par des **algorithmes** très différents
- ▶ On peut (parfois) passer de l'un à l'autre par raffinement
- ▶ L'analyse de **complexité** est un critère fiable pour les comparer
- ▶ ... mais pas le seul

La prochaine fois

- ▶ schémas récursifs
- ▶ analyse du coût en moyenne
- ▶ tri rapide