Introduction à l'informatique

pour les mathématiques, la physique et les sciences computationnelles

Yann Thorimbert

Chapitre 7
Algorithmique, programmation et structures de données

Yann Thorimbert

UNIVERSITÉ DE GENÈVE
CENTRE UNIVERSITAIRE D'INFORMATIQUE

Chapitres du cours

1. Origines des ordinateurs et des réseaux informatiques
2. Codage des nombres
3. Codage des médias
4. Circuits logiques
5. Architecture des ordinateurs
6. Conception et exécution de programmes
7. **Algorithmique, programmation et structures de données** ←

Chapitres du cours (seconde partie du cours)

0. Introduction ←

1. Structures de données : tableaux, listes et dictionnaires

2. Algorithmes de tri

3. Algorithmes de recherche au sein d'une séquence

4. Algorithmes sur graphes

Raisons de concevoir et utiliser une ordinateur

- Automatiser des tâches impliquant un raisonnement logique.

- Comment **décrire** ces tâches logiques ?
  - À un niveau informatique : avec un programme informatique.
  - À un niveau mathématique : avec un **algorithme**.

Qu’est-ce qu’un algorithme ?

- Méthode de résolution systématique d'un problème.
- Sorte de "recette" à appliquer à des données pour résoudre un problème.

Exemple de problème

- Un palindrome est un texte dont la séquence de caractères est la même quel que soit le sens de lecture. Exemple : RADAR est un palindrome.
- On voudrait une méthode systématique pour déterminer si un texte *t* est un palindrome. On cherche un algorithme.
- L'algorithme prend le texte *t* en entrée et retourne une réponse binaire *x* = *vrai* si *t* est un palindrome, *x* = *faux* sinon.
- Choix de notation : on note t_0 la première lettre et t_{n-1} la dernière lettre du texte.

A	M	E	N	A
0	1	2	3	4

Exemple d'algorithme

Pour déterminer si le texte t est un palindrome, voici un pseudo-code :

1. Noter le nombre *n* de caractères au sein de *t*
2. Noter la réponse *x* comme valant *vrai*
3. Pour tous les entiers *i* de 0 à *n-1* :
    a. Noter *opp* = *n* - *i* - 1
    b. Si $t_i \neq t_{opp}$:
        x = *faux*.
        Stopper la boucle
4. La réponse finale est *x.* Fin de l'algorithme.

Exemple d'algorithme

Pour déterminer si le texte t est un palindrome :

1. Noter le nombre *n* de caractères au sein de *t*
2. Noter la réponse *x* comme valant *vrai*
3. Pour tous les entiers *i* de 0 à *n-1* :
   a. Noter *opp* = *n* - *i* - 1
   b. Si $t_i \neq t_{opp}$:
      x = *faux*.
      Stopper la boucle
4. La réponse finale est *x.* Fin de l'algorithme.

On notera : **Retourner** x.

Exemple d'algorithme

Pour déterminer si le texte *t* est un palindrome :

1. Noter le nombre *n* de caractères au sein de *t*
2. Noter la réponse *x* comme valant *vrai*
3. Pour tous les entiers *i* de 0 à *n-1* :
   a. Noter *opp* = *n* - *i* - 1
   b. Si $t_i \neq t_{opp}$:
      x = *faux*.
      Stopper la boucle
4. Retourner x

On peut être plus explicite si nécessaire

Exemple d’algorithme (version détaillée)

Exemple d'algorithme (version détaillée)

Pour déterminer si le texte *t* est un palindrome :

1. Noter le nombre *n* de caractères au sein de *t*
2. Noter la réponse *x* = *vrai*
3. Noter *i* = 0
4. Noter *opp* = *n* - *i* - 1
5. Si $t_i \neq t_{opp}$:
   a. *x* = *faux*
   b. Saut à l'étape 8
6. *i* = *i* + 1
7. Si *i* < *n* :
   a. Saut à l'étape 4
8. Retourner x

Implique une "boucle"

Exemple d'algorithme

Pour déterminer si le texte *t* est un palindrome :

1. Noter le nombre *n* de caractères au sein de *t*
2. Noter la réponse *x* = *vrai*
3. Noter *i* = 0
4. Noter *opp* = *n* - *i* - 1
5. Si $t_i \neq t_{opp}$:
   a. *x* = *faux*
   b. Saut à l'étape 8
6. *i* = *i* + 1
7. Si *i* < *n* :
   a. Saut à l'étape 4
8. Retourner x

Sortie de boucle

Exemple d'algorithme

Application de l'algorithme au texte "NATHAN" :

1. Noter le nombre *n* de caractères au sein de *t*
2. Noter la réponse *x* = *vrai*
3. Noter *i* = 0
4. Noter *opp* = *n* - *i* - 1
5. Si $t_i \neq t_{opp}$:
   a. *x* = *faux*
   b. Saut à l'étape 8
6. *i* = *i* + 1
7. Si *i* < *n* :
   a. Saut à l'étape 4
8. Retourner x

Différentes familles d'algorithmes

- Suivant la façon dont l'algorithme traite le problème, cela peut mener à des nombres d'étapes différents pour le résoudre.
  Par exemple, l'étape 7 de l'algorithme précédent pourrait être : *Si $i < n / 2$*.
- Plus généralement, on peut dégager différentes stratégies qui reviennent souvent en algorithmique.
- Algorithmique : branche à part entière.
- Dans ce cours : un aperçu utile pour les mathématiques et la physique.

Complexité algorithmique

- **Complexité en temps** : notion importante permettant de décrire l'évolution du nombre d'étapes nécessaires pour que l'algorithme se termine en fonction de la taille *n* du problème.
- Pour notre algorithme du palindrome, le nombre d'étapes moyen dans le pire des cas est proportionnel à *n*, la taille du mot. On parle alors de complexité linéaire.
- Nous allons souvent **comparer différents algorithmes** entre eux en nous référant à leur complexité en temps.

Complexité en temps de l'algo du palindrome

Complexité en temps de l'algo du palindrome

1. Noter le nombre *n* de caractères au sein de *t*
2. Noter la réponse *x* = *vrai*
3. Noter *i* = 0
4. Noter *opp* = *n* - *i* - 1
5. Si $t_i \neq t_{opp}$:
   a. *x* = *faux*
   b. Saut à l'étape 8
6. *i* = *i* + 1
7. Si *i* < *n*/2 :
   a. Saut à l'étape 4
8. Retourner x

Effectué une seule fois

Effectué *n* / 2 fois

Complexité en temps de l'algo du palindrome

Il y a 4 étapes qui sont effectuées exactement une fois.

À chacun des n / 2 "tours de boucle", de nombreuses étapes sont effectuées :

- Assignation $opp = n - i - 1$
- Comparaison $t_i \neq t_{opp}$
- Éventuellement : assignation x = faux
- etc.

Admettons que chaque tour de boucle nécessite k étapes en moyenne.

Le nombre d'étapes E pour terminer l'algorithme vaut donc $E = k \cdot n / 2 + 4$

Autrement dit, E est une fonction linéaire de n.

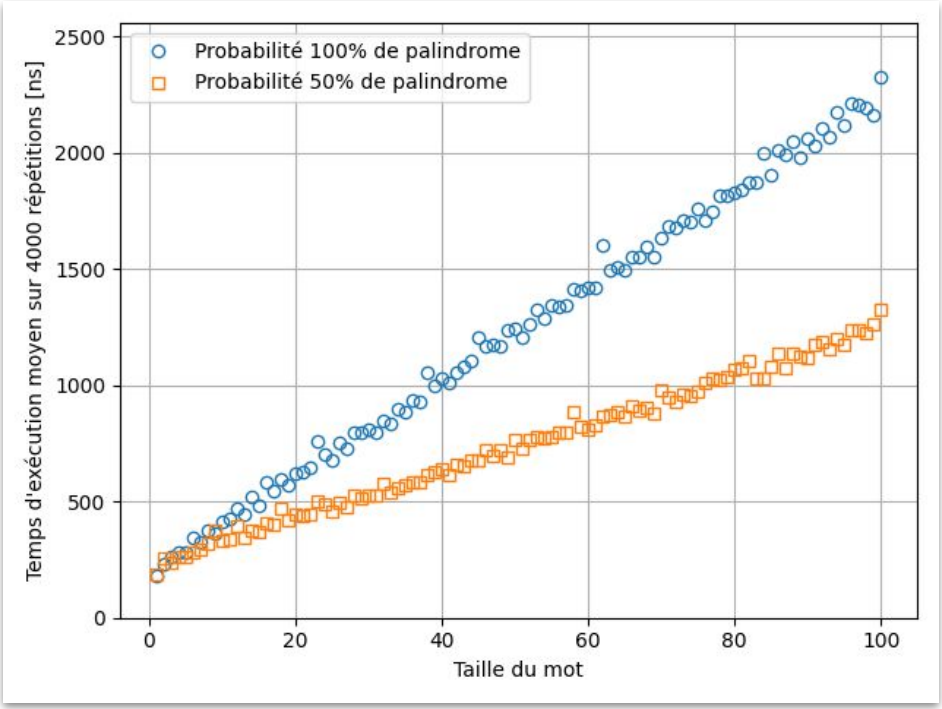
Complexité

- Nous ne sommes **pas intéressés par l'expression exacte** du nombre d'étapes. Ce qui nous intéresse est le **comportement asymptotique** de cette expression, que nous nommerons complexité en temps.
- L'expression exacte *E*(n) du nombre d'étapes, pour une **implémentation** donnée de l'algorithme, est une fonction du type $E(n) = a \cdot n + b$, mais la valeur des constantes dépend de détails d'implémentation et du type d'instructions du processeur.
- Le **temps d'exécution** pour effectuer le nombre d'étapes ajoute une couche de complexité supplémentaire : celle du type exact de hardware utilisé, ainsi que de facteurs externes.
- À un niveau algorithmique, il est pour nous suffisant d'observer que l'algorithme est **linéaire en temps** par rapport à *n*.

Complexité vs temps d'exécution

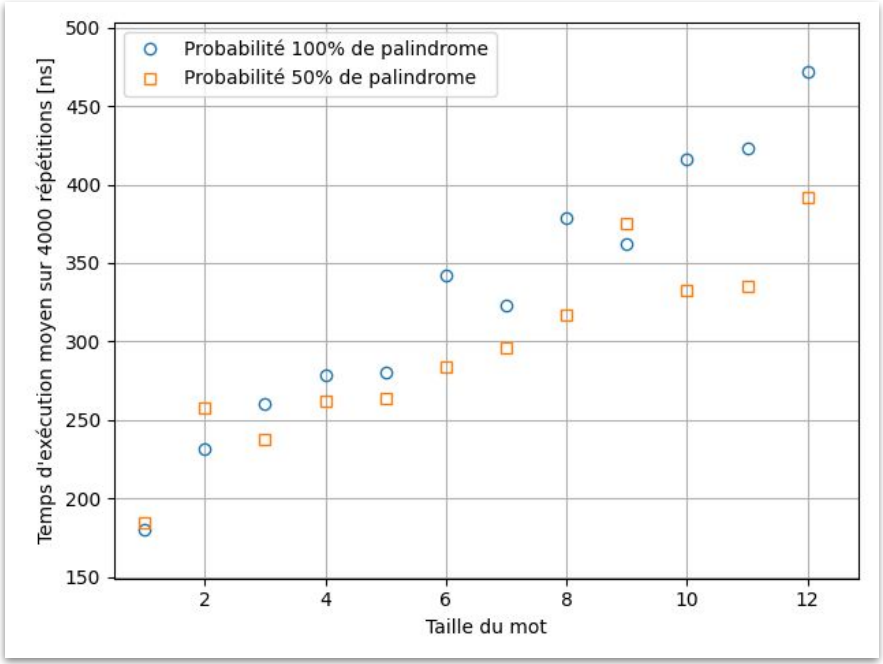
Exemple : chronométrage de la fonction `est_palindrome` implémentée en Python.

```
def est_palindrome(s):
    n = len(s)
    for i in range(n//2):
        if s[i] != s[n-1-i]:
            return False
    return True
```



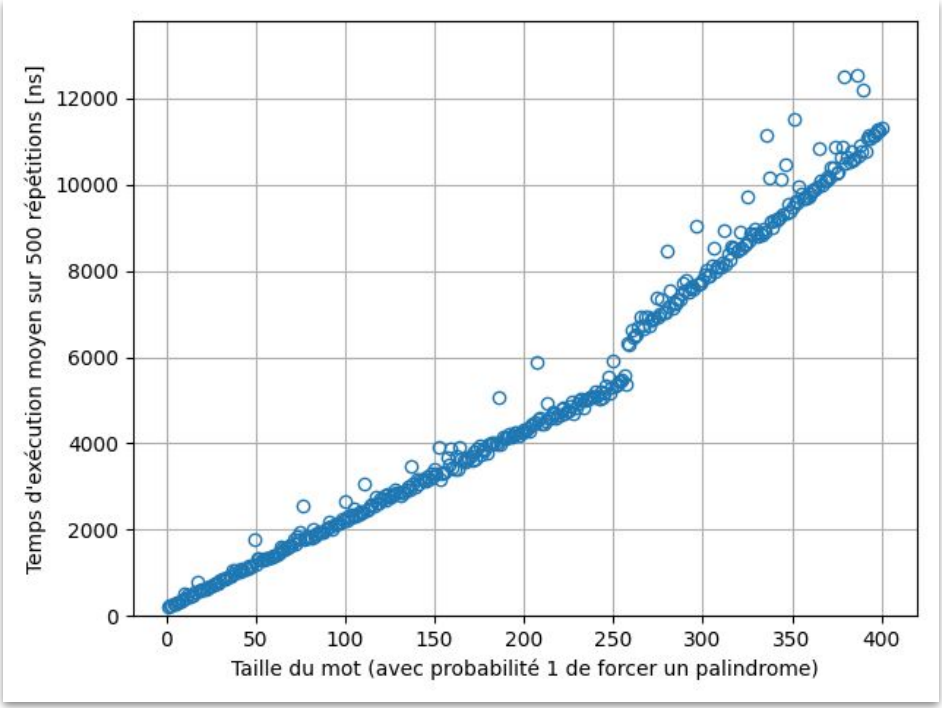
Complexité vs temps d'exécution

- Pour de petites séquences, le temps d'exécution peut parfois être dominé par des effets difficiles à expliquer.
- Explications : changement du flux d'exécution, mauvaises prédiction de branchement (cf. pipelining), optimisations, etc.
- Conclusion : $T(n) \neq a \cdot n + b$. $T(n) = a \cdot n + b + f(n)$, mais $f(n)$ devient négligeable pour n grand.



Complexité vs temps d'exécution

- Au-delà d'une certaine taille, comportement change ⇒ effet de cache ?
- Explications : optimisations, utilisation du cache, etc. (?)
- Conclusion : $T(n) \neq a \cdot n + b$. $T(n) = a \cdot n + b + f(n)$, mais $f(n)$ devient négligeable pour n grand.



Complexité | **Commentaires**

Complexité | **Commentaires**

Pour déterminer la complexité en temps :

A. Exprimer le temps de calcul T(n), par exemple :
   $T(n) = 4n^2 + 3n + 25 \log(n) + 12$

B. Garder le terme dominant pour n très grand : $T(n) \approx 4n^2$

C. Ôter les facteurs constants : $T(n) = O(n^2)$

Cf. démonstration pour le comptage des opérations dans une boucle multiple.

Complexité | **Définitions formelles**

L'analyse de complexité peut être plus précise et poussée, mais ce n'est pas le sujet du cours. Voici néanmoins des définitions que l'on trouve fréquemment dans la littérature :

$T(n) = O(g(n))$ dénote en général le pire des cas : $\exists\ k, n_0 \in \mathbb{R}_+ \mid T(n) \leq k \cdot g(n),\ \forall\ n > n_0$
(autrement dit, "T est **tout au plus** d'une complexité g(n)")

$T(n) = \Theta(g(n))$ signifie que $\exists\ a, b, n_0 \in \mathbb{R}_+ \mid a \cdot g(n) \leq T(n) \leq b \cdot g(n),\ \forall\ n > n_0$
(autrement dit, "T est d'une complexité g(n)")

(Hors champ)

Types courants de complexités

Nom croissance	Complexité en temps	Exemples d'algorithmes
Constante	$\mathcal{O}(1)$	Trouver si un nombre binaire est pair
Logarithmique	$\mathcal{O}(\log(n))$	Recherche dichotomique
Linéaire	$\mathcal{O}(n)$	Trouver le minimum d'une liste
Linéarithmique	$\mathcal{O}(n \log(n))$	Tri fusion
Quadratique	$\mathcal{O}(n^2)$	Calculer la couleur moyenne d'une image
Cubique	$\mathcal{O}(n^3)$	Multiplication naïve de matrices $n \times n$
Exponentielle	$\mathcal{O}(2^n)$	Casser un mot de passe par force brute
Factorielle	$\mathcal{O}(n!)$ ou $\mathcal{O}(n^n)$	Générer toutes les permutations d'un ensemble

Complexité et structures de données

- Pour décrire un algorithme comme pour l'implémenter, il est souvent pratique de prendre pour acquis une façon dont les données sont structurées.

- Exemple : dans le palindrome, on a implicitement supposé que les caractères d'un mot étaient ordonnés (structure de donnée : tableau de caractères) et accessibles via leur "numéro" (ou indice).

- Certaines structures reviennent très souvent :
  - Tableaux
  - Listes
  - Arbres
  - …

Seront étudié dans les prochains chapitres en même temps que certains algorithmes.

Complexité | **Commentaires**

- Il faut noter que l'algorithme présenté est $O(n)$ seulement si l'accès aux éléments de la chaîne de caractères est $O(1)$.

- Dans le cas contraire, il faut en tenir compte ! Par exemple, si la complexité moyenne d'accès aux éléments est $O(n^3)$, alors l'algorithme dans son ensemble est $O(n^5)$.

- On peut s'intéresser à d'autres complexités que la complexité en temps, comme la complexité en mémoire. Comme ce n'est pas le cas dans ce cours, nous omettons souvent de préciser qu'il s'agit de la complexité en temps.

Commentaires finaux sur la complexité

Dans certains cas, une complexité élevée peut être désirable. Exemple lié à l'authentification :

- On choisit deux nombres premiers a et b et on calcule c = a · b. Le nombre c admet donc deux diviseurs.
- Un site stocke la valeur c associée à notre compte. Pour prouver notre identité, nous devons donner un diviseur de c. Le site peut facilement vérifier que c est divisible par le nombre qu'on lui donne (a ou b), en effectuant la division.
- Si c possède k chiffres décimaux, un attaquant doit quant à lui vérifier $O(10^{k-1})$ nombres (avec une approche naïve). Avec une approche non-naïve, la complexité reste quasiment exponentielle.
- Beaucoup d'algorithmes de cryptographie fonctionnent sur ce principe, où tout repose sur la difficulté pratique à résoudre un problème de factorisation (et la facilité à vérifier la réponse).

(Hors champ)

Temps d'exécution curieux

- Créons artificiellement une fonction avec un grand overhead : la fonction `f(n)` doit retourner une liste contenant n fois la valeur k, c'est-à-dire `[k, k,... k]`.
- La valeur k est calculée en fonction de la valeur de n comme :

$$k(n) = \sum_{i=1}^{N=100} \left[\frac{-1}{n + 0.1} \right]^i$$

- La complexité du calcul de k(n) est constante (environ 200 additions, 100 divisions, 100 calculs de puissance).
- La complexité de l'algorithme est donc O(n)...
- … quid du temps d'exécution ? Dépendances cachées en n.

(Hors champ)