Algorithmique et structure de données

Complexité, Récursivité et tableaux

Rémi Giraud

2024-2025

Slides conçus grâce aux supports de Rohan Fossé

Objectifs

- Se familiariser avec des problèmes classiques et leur solutions ;
- Se préparer à trouver des solutions algorithmiques à des problèmes en sachant comparer leurs performances ;
- S'entraîner à écrire des algorithmes et connaître les différentes structures de données.

Table des matières

Introduction

Algorithmes

Exemple d'algorithme

Se laver les mains

1. Ouvrir l'eau

Exemple d'algorithme

Se laver les mains

1. Ouvrir l'eau
2. Mettre du savon

Exemple d'algorithme

Se laver les mains

1. Ouvrir l'eau
2. Mettre du savon
3. Nettoyer ses mains avec l'eau

Exemple d'algorithme

Se laver les mains

1. Ouvrir l'eau
2. Mettre du savon
3. Nettoyer ses mains avec l'eau
4. Éteindre l'eau

Exemple d'algorithme

Se laver les mains

1. Ouvrir l'eau
2. Mettre du savon
3. Nettoyer ses mains avec l'eau
4. Éteindre l'eau
5. Se sécher les mains

Autre exemple d'algorithme

Boire son café

1. Prenez une dosette de café.

Autre exemple d'algorithme

Boire son café

1. Prenez une dosette de café.
2. Mettez-la dans la machine à café.

Autre exemple d'algorithme

Boire son café

1. Prenez une dosette de café.
2. Mettez-la dans la machine à café.
3. Vérifiez si la machine à café est allumée. Si ce n'est pas le cas, allumez la machine.

Autre exemple d'algorithme

Boire son café

1. Prenez une dosette de café.
2. Mettez-la dans la machine à café.
3. Vérifiez si la machine à café est allumée. Si ce n'est pas le cas, allumez la machine.
4. Vérifiez si le filtre à eau est suffisamment rempli. - Si ce n'est pas le cas, ajoutez de l'eau.

Autre exemple d'algorithme

Boire son café

1. Prenez une dosette de café.
2. Mettez-la dans la machine à café.
3. Vérifiez si la machine à café est allumée. Si ce n'est pas le cas, allumez la machine.
4. Vérifiez si le filtre à eau est suffisamment rempli. - Si ce n'est pas le cas, ajoutez de l'eau.
5. Placez une tasse à café sous le distributeur de café.

Autre exemple d'algorithme

Boire son café

1. Prenez une dosette de café.
2. Mettez-la dans la machine à café.
3. Vérifiez si la machine à café est allumée. Si ce n'est pas le cas, allumez la machine.
4. Vérifiez si le filtre à eau est suffisamment rempli. - Si ce n'est pas le cas, ajoutez de l'eau.
5. Placez une tasse à café sous le distributeur de café.
6. Appuyez sur le bouton café.

Autre exemple d'algorithme

Boire son café

1. Prenez une dosette de café.
2. Mettez-la dans la machine à café.
3. Vérifiez si la machine à café est allumée. Si ce n'est pas le cas, allumez la machine.
4. Vérifiez si le filtre à eau est suffisamment rempli. - Si ce n'est pas le cas, ajoutez de l'eau.
5. Placez une tasse à café sous le distributeur de café.
6. Appuyez sur le bouton café.
7. Le café est en train d'être servi. - Attendez que la machine indique que le café est prêt.

Autre exemple d'algorithme

Boire son café

1. Prenez une dosette de café.
2. Mettez-la dans la machine à café.
3. Vérifiez si la machine à café est allumée. Si ce n'est pas le cas, allumez la machine.
4. Vérifiez si le filtre à eau est suffisamment rempli. - Si ce n'est pas le cas, ajoutez de l'eau.
5. Placez une tasse à café sous le distributeur de café.
6. Appuyez sur le bouton café.
7. Le café est en train d'être servi. - Attendez que la machine indique que le café est prêt.
8. Si le café est prêt - Sortez la tasse à café de la machine à café.

Autre exemple d'algorithme

Boire son café

1. Prenez une dosette de café.
2. Mettez-la dans la machine à café.
3. Vérifiez si la machine à café est allumée. Si ce n'est pas le cas, allumez la machine.
4. Vérifiez si le filtre à eau est suffisamment rempli. - Si ce n'est pas le cas, ajoutez de l'eau.
5. Placez une tasse à café sous le distributeur de café.
6. Appuyez sur le bouton café.
7. Le café est en train d'être servi. - Attendez que la machine indique que le café est prêt.
8. Si le café est prêt - Sortez la tasse à café de la machine à café.

Qu'est ce qu'un algorithme ?

Un algorithme

- Un algorithme est une méthode systématique (comme une recette) pour résoudre un problème donné ;
- Il se compose d'une suite d'opérations simples à effectuer pour résoudre ce problème ;
- En informatique, **cette méthode doit être applicable par un ordinateur**.
  - **Algorithme** : séquence d'opérations de calcul élémentaires, organisée selon des règles précises dans le but de résoudre un problème donné.
  - **Structures de données** : moyen de stocker et organiser des données pour faciliter l'accès à ces données et leur modification.

Importance de l'algorithmique

- Pour un problème donné, il existe plusieurs algorithmes ;
- Il est facile d'écrire des algorithmes faux ou inefficaces ;
- Un mauvais choix ou une erreur peut faire la différence entre quelques minutes de calculs et plusieurs heures ;
- C'est souvent une question d'utilisation de structures de données ou d'algorithmes connus dans la littérature.

Exemple : la ville et la pizzeria

Le problème

Nous allons considérer 3 villes contenant 14 maisons et 1 pizzeria. Nous souhaitons savoir quelle ville permet aux livreurs de pizza de faire les trajets les plus rapides et pourquoi.

Figure 1 – 1 pizzeria et 14 maisons

Exemple : la ville et la pizzeria

Le problème

Nous allons considérer 3 villes contenant 14 maisons et 1 pizzeria. Nous souhaitons savoir quelle ville permet aux livreurs de pizza de faire les trajets les plus rapides et pourquoi.

Temps de calcul

On suppose qu'il faut une unité de temps pour passer d'une maison à un autre, en suivant une rue.

Figure 2 – Trajet entre la pizzeria et une maison

Exemple : la ville et la pizzeria

Le problème

Nous allons considérer 3 villes contenant 14 maisons et 1 pizzeria. Nous souhaitons savoir quelle ville permet aux livreurs de pizza de faire les trajets les plus rapides et pourquoi.

Temps de calcul

On suppose qu'il faut une unité de temps pour passer d'une maison à un autre, en suivant une rue.

Figure 2 – Trajet entre la pizzeria et une maison

Dans le pire cas, quel est le temps mis par un livreur pour aller de la pizzeria jusqu'à une maison ?

Ville A

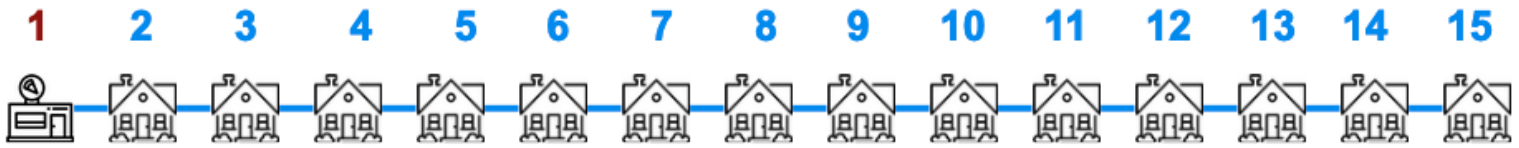
Organisation de la ville

Les maisons sont rangées dans l'ordre croissant en ligne droite. La pizzeria se trouve au numéro 1.

Ville A

Organisation de la ville

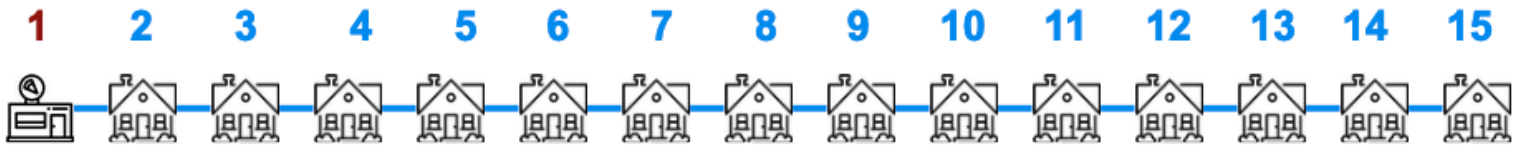
Les maisons sont rangées dans l'ordre croissant en ligne droite. La pizzeria se trouve au numéro 1.



Ville A

Organisation de la ville

Les maisons sont rangées dans l'ordre croissant en ligne droite. La pizzeria se trouve au numéro 1.

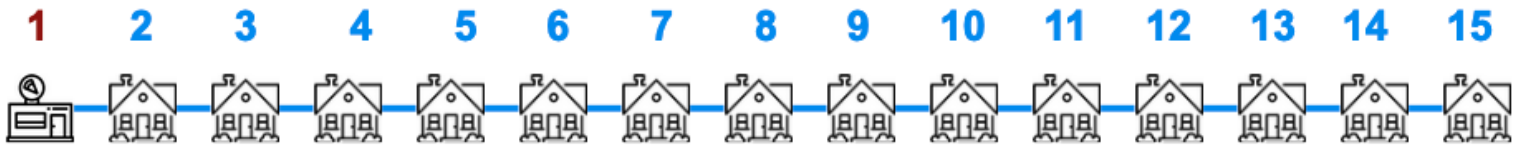


Quel est le pire temps possible ?

Ville A

Organisation de la ville

Les maisons sont rangées dans l'ordre croissant en ligne droite. La pizzeria se trouve au numéro 1.



Quel est le pire temps possible ? **14**

Ville B

Organisation de la ville

Même organisation que dans la ville A, mais cette fois-ci la pizzeria est au numéro **8**

Ville B

Organisation de la ville

Même organisation que dans la ville A, mais cette fois-ci la pizzeria est au numéro **8**

Ville B

Organisation de la ville

Même organisation que dans la ville A, mais cette fois-ci la pizzeria est au numéro **8**

Quel est le pire temps possible ?

Ville B

Organisation de la ville

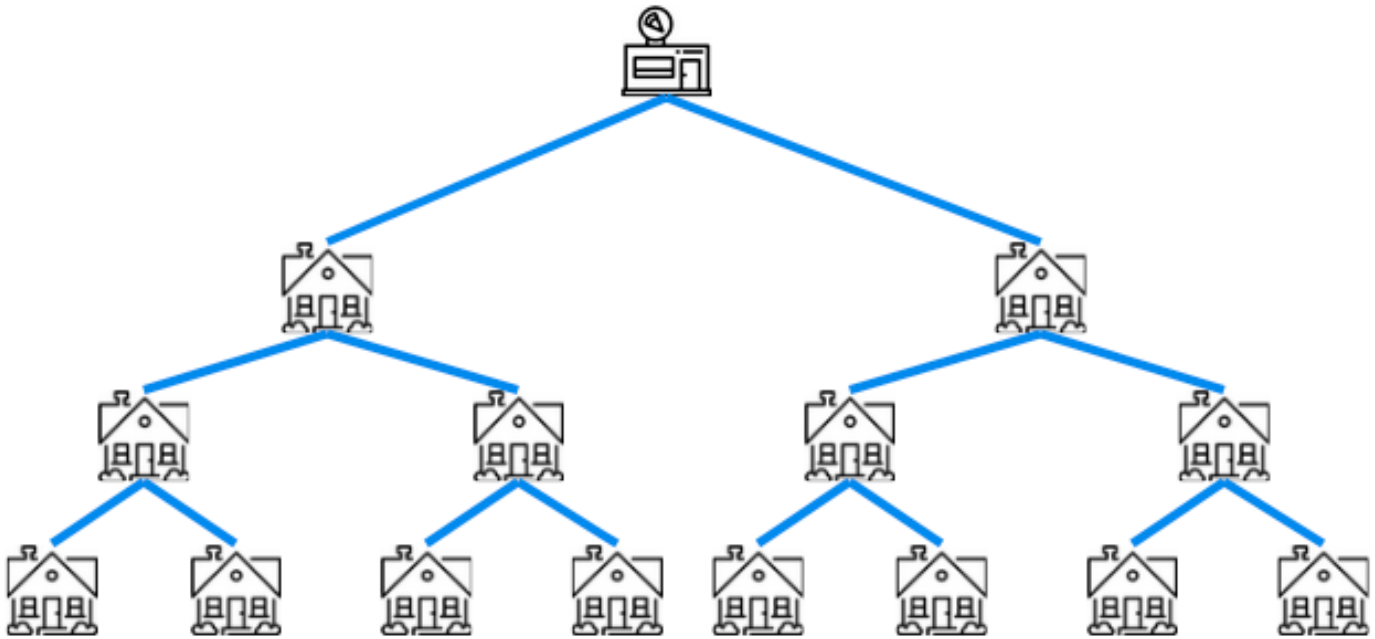
Même organisation que dans la ville A, mais cette fois-ci la pizzeria est au numéro **8**

Quel est le pire temps possible ? **7**

Ville C

Organisation de la ville

Cette fois-ci, les maisons sont organisées en embranchements, la pizzeria est tout en **haut**.

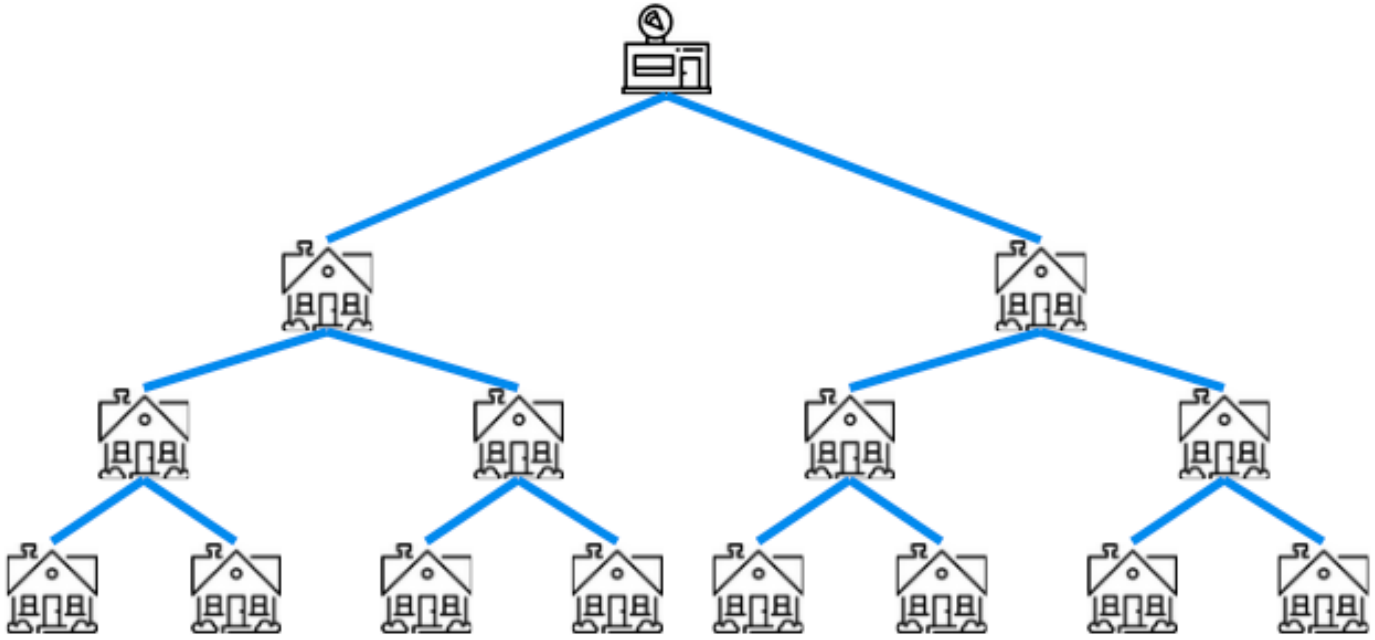


Quel est le pire temps possible ?

Ville C

Organisation de la ville

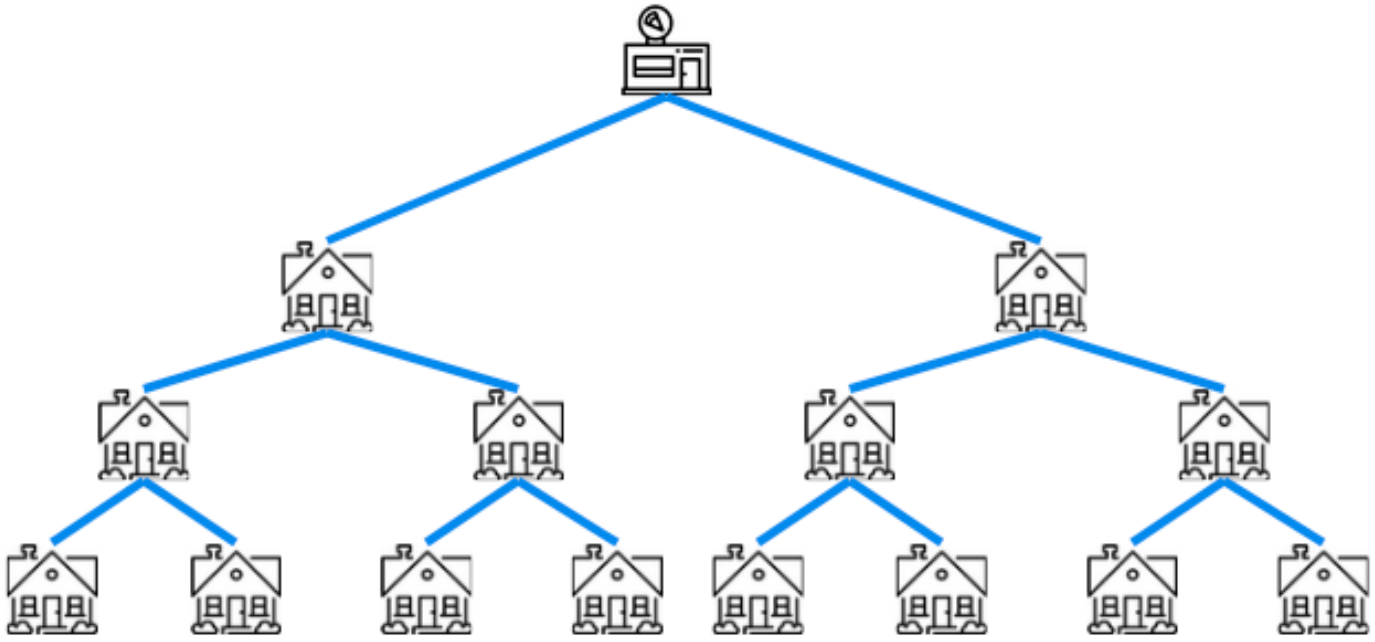
Cette fois-ci, les maisons sont organisées en embranchements, la pizzeria est tout en **haut**.



Quel est le pire temps possible ? **3**

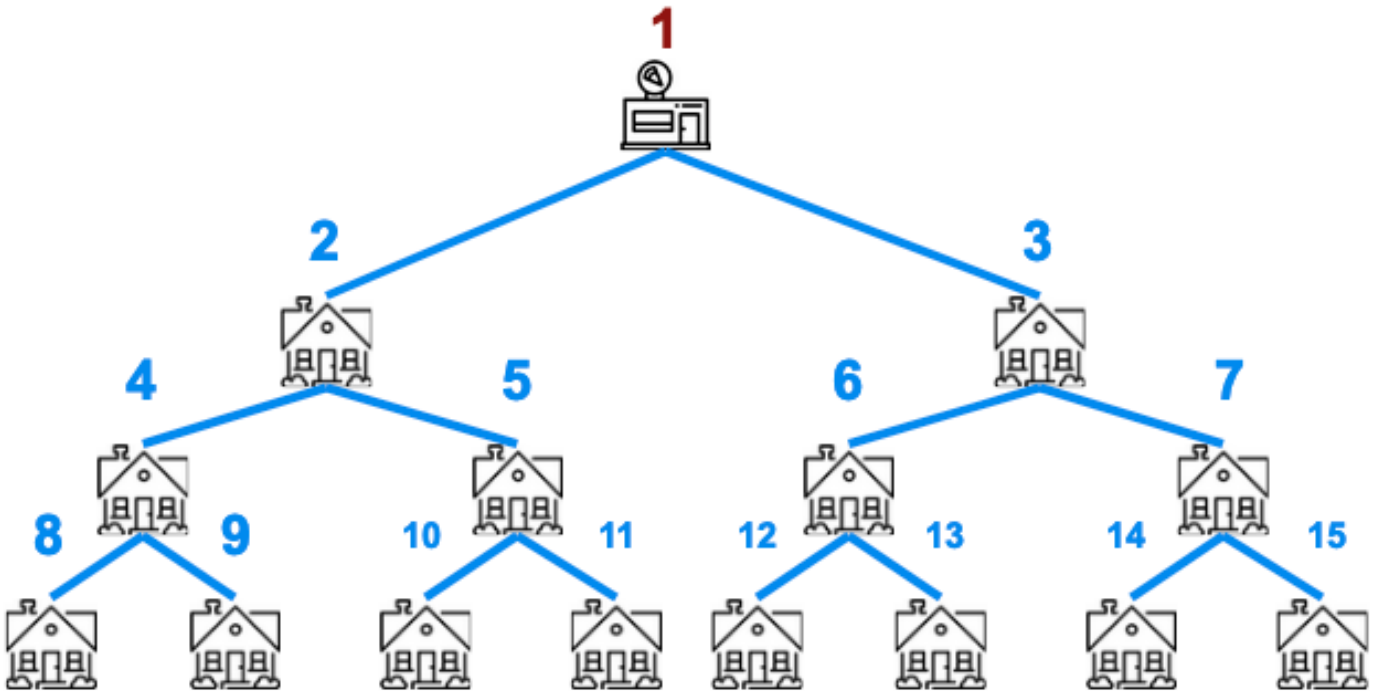
Organisation de la ville

Cette fois-ci, les maisons sont organisées en embranchements, la pizzeria est tout en **haut**.

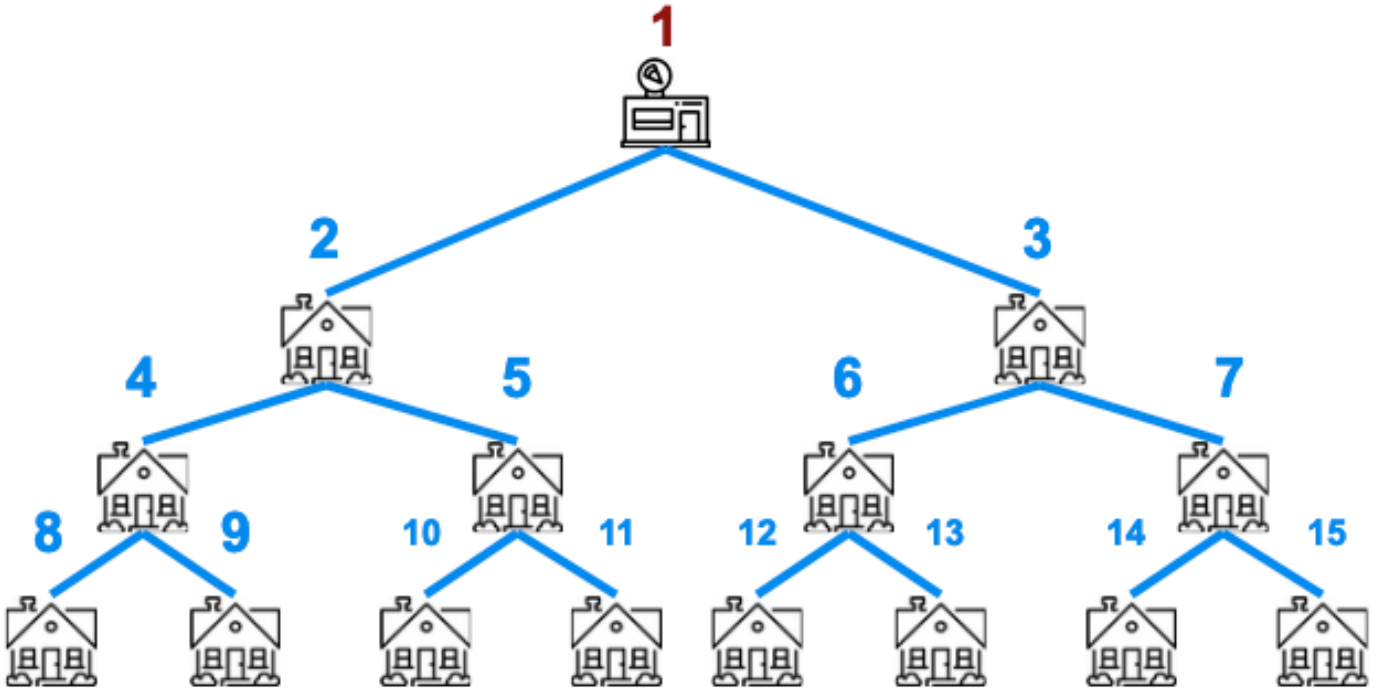


Comment numéroter les maisons ?

Ville C : première numérotation

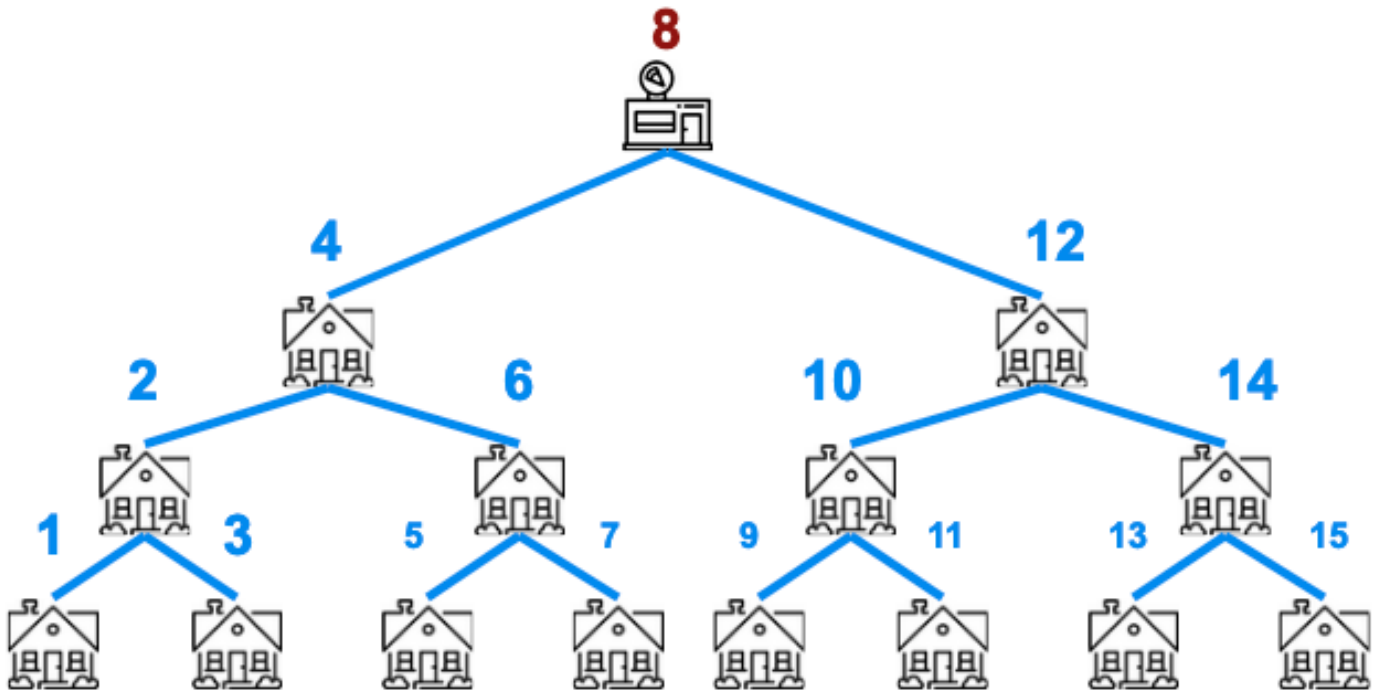


Ville C : première numérotation

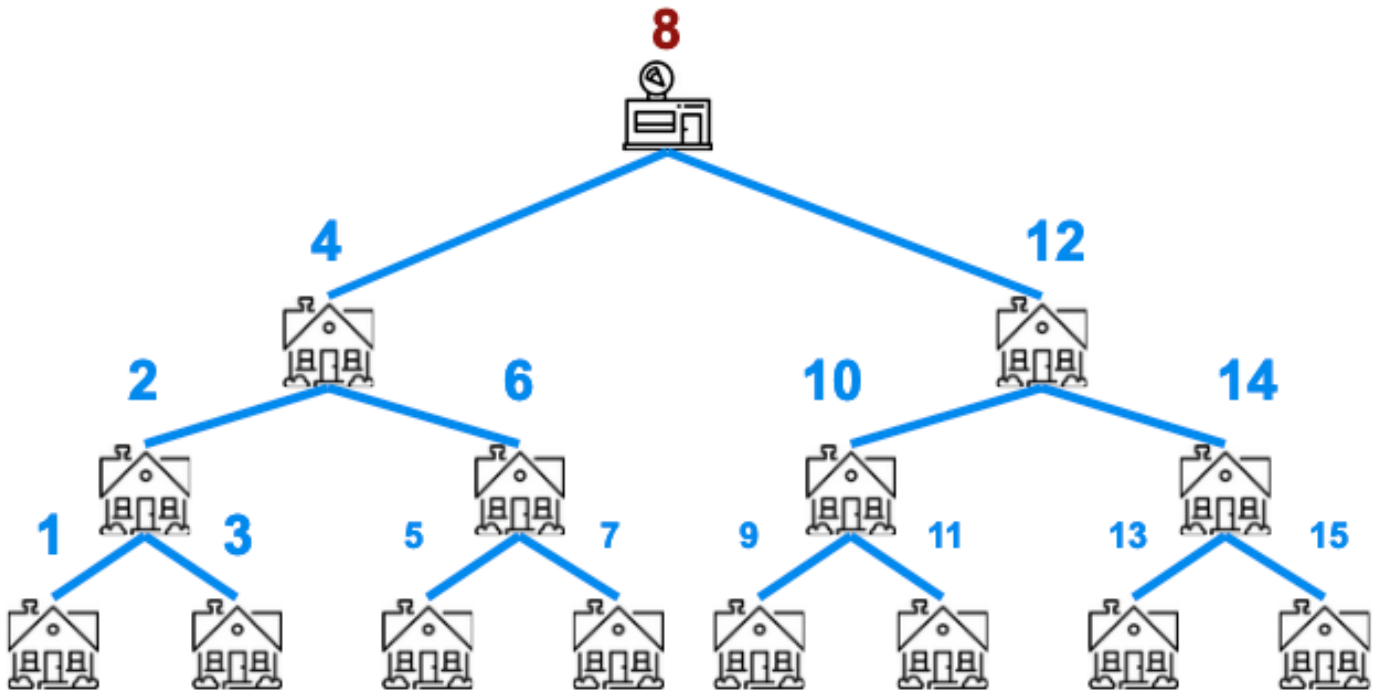


Comment choisir la bonne rue à prendre ?

Ville C : deuxième numérotation



Ville C : deuxième numérotation



Dans ce cas là, si le numéro de la maison à livrer est plus petit que celui sur lequel on se trouve, on va à gauche, sinon on va à droite.

Tableau récapitulatif

Nombre de maisons	Ville A	Ville B	Ville C
15	14	7	3

Tableau récapitulatif

Nombre de maisons	**Ville A**	**Ville B**	**Ville C**
15	14	7	3
1023			

Tableau récapitulatif

Nombre de maisons	**Ville A**	**Ville B**	**Ville C**
15	14	7	3
1023	1022	511	9

Tableau récapitulatif

Nombre de maisons	**Ville A**	**Ville B**	**Ville C**
15	14	7	3
1023	1022	511	9
n			

Tableau récapitulatif

Nombre de maisons	Ville A	Ville B	Ville C
15	14	7	3
1023	1022	511	9
n	n-1		

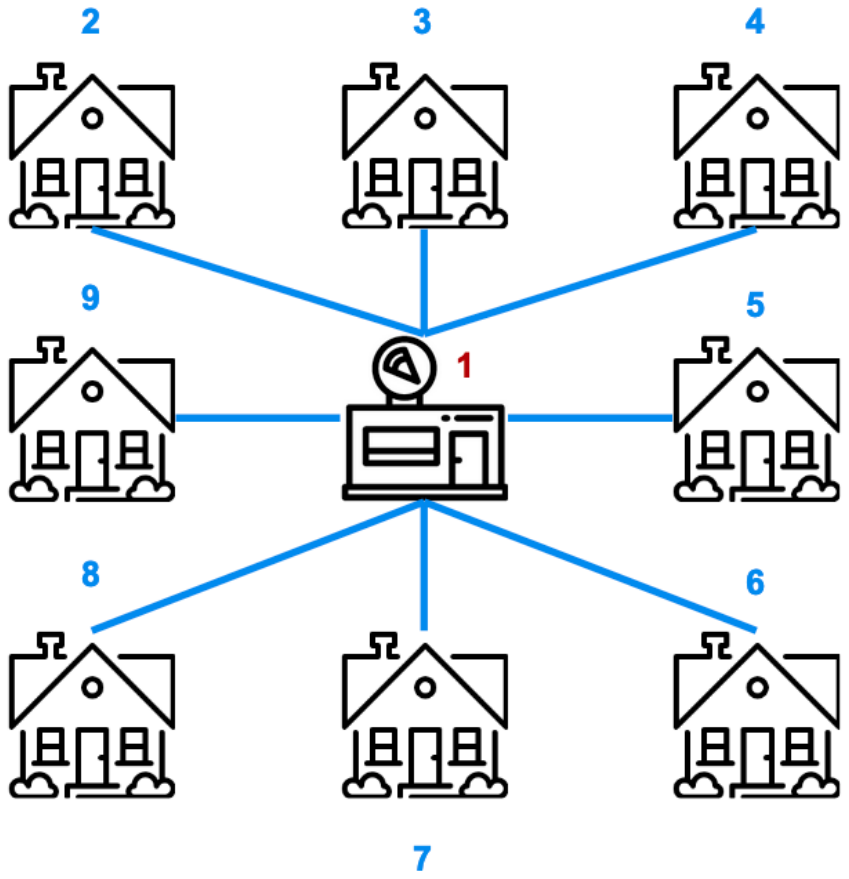
Tableau récapitulatif

Nombre de maisons	**Ville A**	**Ville B**	**Ville C**
15	14	7	3
1023	1022	511	9
n	n-1	$\frac{n-1}{2}$	

Tableau récapitulatif

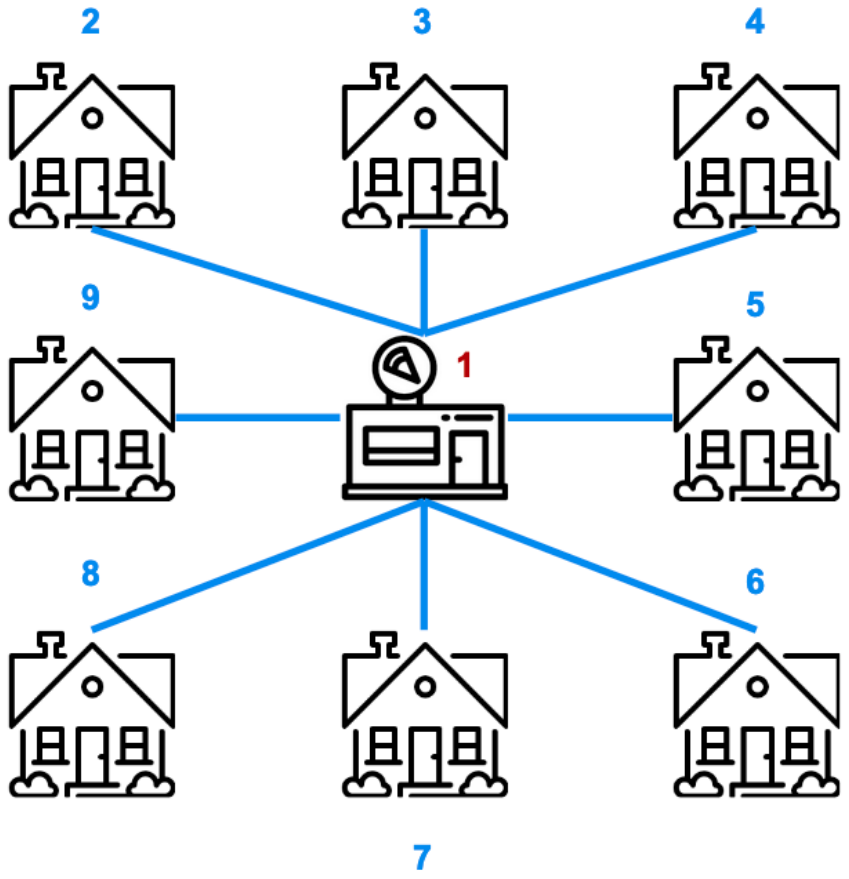
Nombre de maisons	Ville A	Ville B	Ville C
15	14	7	3
1023	1022	511	9
n	n-1	$\frac{n-1}{2}$	$\approx log_2(n)$

Et pourquoi pas en étoile ?



Cette configuration semble optimale, quel peut être le problème ?

Et pourquoi pas en étoile ?



Une organisation en étoile avec la pizzeria au milieu permet des trajets très courts, mais choisir la bonne rue prend du temps.

Complexité : quelques définitions

Complexité

La **complexité** dans le pire cas d'un algorithme est la fonction mathématique $\mathcal{T}$ qui donne le nombre maximal d'instructions élémentaires que l'algorithme effectue en fonction de la taille des données manipulées.

Exemple précédent

Dans notre exemple précédent, la taille des données manipulées étaient le nombre de maisons.

Complexité dans le pire cas : À quoi ça sert ?

- **Évaluer le temps d'exécution** d'un algorithme en fonction de la longueur de l'entrée ;
- **Comparer les performances** de différents algorithmes résolvant le même problème ;
- **Évaluer la taille maximale** des entrées qu'un algorithme peut traiter ;
- Mesure du temps **indépendante des machines** puisque l'on compte le nombre d'instructions.

Ordre de grandeur

Pour comparer des algorithmes, il n'est pas nécessaire d'utiliser la fonction $\mathcal{T}$, mais seulement l'ordre de grandeur asymptotique, noté $\mathcal{O}$ (prononcé "*grand O*").

Définition formelle

Une fonction $\mathcal{T}(n)$ est en $\mathcal{O}(f(n))$ ("*en grand O de f(n)*") si :

$$\exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}^+, \forall n \in \mathbb{R}^+, n \geq n_0 \implies |T(n)| \leq c|f(n)|$$

Autrement dit, $\mathcal{T}(n)$ est en $\mathcal{O}(f(n))$ s'il existe un seuil n_0 à partir duquel la fonction $\mathcal{T}$ est toujours dominée par la fonction f, à une constance multiplicative fixée c.

Quelques exemples

Exemples

- $T_1(n) = 7 =$
- $T_2(n) = 12n + 5$
- $T_3(n) = 4n^2 + 2n + 6$
- $T_4(n) = 2 + (n - 1) \times 5$

Quelques exemples

Exemples

- $T_1(n) = 7 = \mathcal{O}(7) =$
- $T_2(n) = 12n + 5$
- $T_3(n) = 4n^2 + 2n + 6$
- $T_4(n) = 2 + (n - 1) \times 5$

Quelques exemples

Exemples

- $T_1(n) = 7 = \mathcal{O}(7) = \mathcal{O}(1)$
- $T_2(n) = 12n + 5$
- $T_3(n) = 4n^2 + 2n + 6$
- $T_4(n) = 2 + (n - 1) \times 5$

Quelques exemples

Exemples

- $T_1(n) = 7 = \mathcal{O}(7) = \mathcal{O}(1)$
- $T_2(n) = 12n + 5 = \mathcal{O}(12n)$
- $T_3(n) = 4n^2 + 2n + 6$
- $T_4(n) = 2 + (n - 1) \times 5$

Quelques exemples

Exemples

- $T_1(n) = 7 = \mathcal{O}(7) = \mathcal{O}(1)$
- $T_2(n) = 12n + 5 = \mathcal{O}(12n) = \mathcal{O}(n)$
- $T_3(n) = 4n^2 + 2n + 6$
- $T_4(n) = 2 + (n - 1) \times 5$

Quelques exemples

Exemples

- $T_1(n) = 7 = \mathcal{O}(7) = \mathcal{O}(1)$
- $T_2(n) = 12n + 5 = \mathcal{O}(12n) = \mathcal{O}(n)$
- $T_3(n) = 4n^2 + 2n + 6 = \mathcal{O}(4n^2 + 2n)$
- $T_4(n) = 2 + (n - 1) \times 5$

Quelques exemples

Exemples

- $T_1(n) = 7 = \mathcal{O}(7) = \mathcal{O}(1)$
- $T_2(n) = 12n + 5 = \mathcal{O}(12n) = \mathcal{O}(n)$
- $T_3(n) = 4n^2 + 2n + 6 = \mathcal{O}(4n^2 + 2n) = \mathcal{O}(n^2)$
- $T_4(n) = 2 + (n - 1) \times 5$

Quelques exemples

Exemples

- $T_1(n) = 7 = \mathcal{O}(7) = \mathcal{O}(1)$
- $T_2(n) = 12n + 5 = \mathcal{O}(12n) = \mathcal{O}(n)$
- $T_3(n) = 4n^2 + 2n + 6 = \mathcal{O}(4n^2 + 2n) = \mathcal{O}(n^2)$
- $T_4(n) = 2 + (n - 1) \times 5 = \mathcal{O}(n - 1)$

Quelques exemples

Exemples

- $T_1(n) = 7 = \mathcal{O}(7) = \mathcal{O}(1)$
- $T_2(n) = 12n + 5 = \mathcal{O}(12n) = \mathcal{O}(n)$
- $T_3(n) = 4n^2 + 2n + 6 = \mathcal{O}(4n^2 + 2n) = \mathcal{O}(n^2)$
- $T_4(n) = 2 + (n - 1) \times 5 = \mathcal{O}(n - 1) = \mathcal{O}(n)$

Quelques exemples

Exemples

- $T_1(n) = 7 = \mathcal{O}(7) = \mathcal{O}(1)$
- $T_2(n) = 12n + 5 = \mathcal{O}(12n) = \mathcal{O}(n)$
- $T_3(n) = 4n^2 + 2n + 6 = \mathcal{O}(4n^2 + 2n) = \mathcal{O}(n^2)$
- $T_4(n) = 2 + (n-1) \times 5 = \mathcal{O}(n-1) = \mathcal{O}(n)$

À vous de jouer

Quelle est la complexité dans le pire des cas des exemples suivants ?

- $T_5(n) = 1024$
- $T_6(n) = 3n + log(n)^4 + 2$
- $T_7(n) = 2^n + n^{10}$

Quelques exemples

Exemples

- $T_1(n) = 7 = \mathcal{O}(7) = \mathcal{O}(1)$
- $T_2(n) = 12n + 5 = \mathcal{O}(12n) = \mathcal{O}(n)$
- $T_3(n) = 4n^2 + 2n + 6 = \mathcal{O}(4n^2 + 2n) = \mathcal{O}(n^2)$
- $T_4(n) = 2 + (n-1) \times 5 = \mathcal{O}(n-1) = \mathcal{O}(n)$

À vous de jouer

Quelle est la complexité dans le pire des cas des exemples suivants ?

- $T_5(n) = 1024 = \mathcal{O}(1)$
- $T_6(n) = 3n + log(n)^4 + 2$
- $T_7(n) = 2^n + n^{10}$

Quelques exemples

Exemples

- $T_1(n) = 7 = \mathcal{O}(7) = \mathcal{O}(1)$
- $T_2(n) = 12n + 5 = \mathcal{O}(12n) = \mathcal{O}(n)$
- $T_3(n) = 4n^2 + 2n + 6 = \mathcal{O}(4n^2 + 2n) = \mathcal{O}(n^2)$
- $T_4(n) = 2 + (n-1) \times 5 = \mathcal{O}(n-1) = \mathcal{O}(n)$

À vous de jouer

Quelle est la complexité dans le pire des cas des exemples suivants ?

- $T_5(n) = 1024 = \mathcal{O}(1)$
- $T_6(n) = 3n + log(n)^4 + 2 = \mathcal{O}(n)$
- $T_7(n) = 2^n + n^{10}$

Quelques exemples

Exemples

- $T_1(n) = 7 = \mathcal{O}(7) = \mathcal{O}(1)$
- $T_2(n) = 12n + 5 = \mathcal{O}(12n) = \mathcal{O}(n)$
- $T_3(n) = 4n^2 + 2n + 6 = \mathcal{O}(4n^2 + 2n) = \mathcal{O}(n^2)$
- $T_4(n) = 2 + (n-1) \times 5 = \mathcal{O}(n-1) = \mathcal{O}(n)$

À vous de jouer

Quelle est la complexité dans le pire des cas des exemples suivants ?

- $T_5(n) = 1024 = \mathcal{O}(1)$
- $T_6(n) = 3n + log(n)^4 + 2 = \mathcal{O}(n)$
- $T_7(n) = 2^n + n^{10} = \mathcal{O}(2^n)$

Rappels des règles utiles

Les quelques règles suivantes permettent de simplifier les complexités en omettant des termes dominés :

- Les coefficients peuvent être omis : $14n^2$ devient n^2 ;
- n^a domine n^b si $a > b$: par exemple, n^2 domine n ;
- Une exponentielle domine un polynôme : $3^n = exp^{nlog3}$ domine n^5 ;
- De même un polynôme domine un logarithme : n domine $(logn)^3$. Cela signifie également que, par exemple, n^2 domine $nlogn$.

Reprenons notre pizzeria

Nombre de maisons	Ville A	Ville B	Ville C
15	14	7	3
1023	1022	511	9
n	n-1	$\frac{n-1}{2}$	$log_2(n)$

Reprenons notre pizzeria

Nombre de maisons	Ville A	Ville B	Ville C
15	14	7	3
1023	1022	511	9
n	n-1	$\frac{n-1}{2}$	$log_2(n)$
Complexité pour n	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(log(n))$

Reprenons notre pizzeria

Nombre de maisons	Ville A	Ville B	Ville C
15	14	7	3
1023	1022	511	9
n	n-1	$\frac{n-1}{2}$	$log_2(n)$
Complexité pour n	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(log(n))$

On voit que les complexités dans le pire cas de la ville A et de la ville B **sont les mêmes**

Reprenons notre exemple

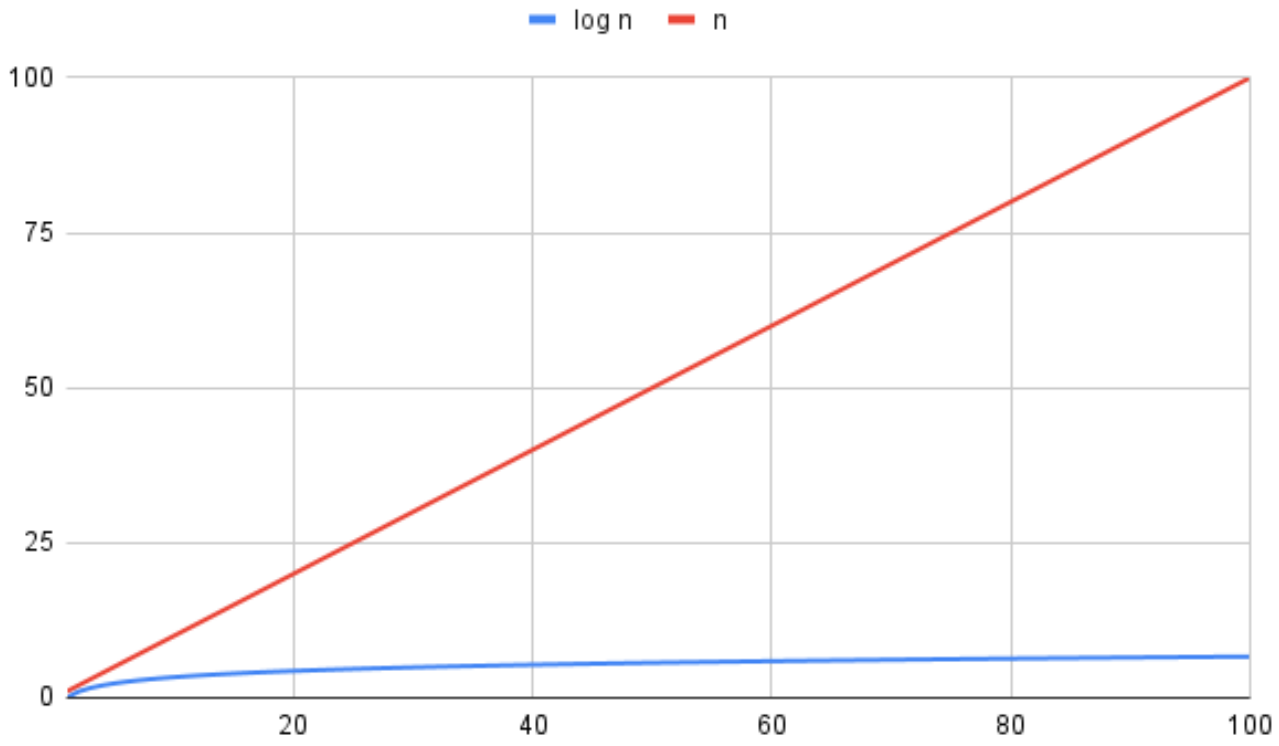
- Dans la ville A et B, l'algorithme naturel pour trouver une maison a une complexité linéaire $\mathcal{O}(n)$
- Dans la ville C, l'algorithme naturel pour trouver une maison a une complexité logarithmique $\mathcal{O}(log(n))$
- L'organisation de la ville C est beaucoup plus efficace pour livrer les pizzas.

Reprenons notre exemple

- Dans la ville A et B, l'algorithme naturel pour trouver une maison a une complexité linéaire $\mathcal{O}(n)$
- Dans la ville C, l'algorithme naturel pour trouver une maison a une complexité logarithmique $\mathcal{O}(log(n))$
- L'organisation de la ville C est beaucoup plus efficace pour livrer les pizzas.

À quel point est-ce que c'est plus efficace ?

Différence entre n et log n



- Si $n = 10^6$, alors $log_2(n) \approx 20$. Le livreur fait 50 000 fois moins de déplacements si les maisons sont organisées comme dans la ville C.

Classe de complexité

Complexité	Notation
constante	$\mathcal{O}(1)$
logarithmique	$\mathcal{O}(log(n))$
racinaire	$\mathcal{O}(\sqrt{x})$
linéaire	$\mathcal{O}(n)$
quasi-linéaire	$\mathcal{O}(nlog(n))$
quadratique (polynomial)	$\mathcal{O}(n^2)$
cubique (polynomial)	$\mathcal{O}(n^3)$
sous-exponentielle	$\mathcal{O}(2^{poly(log(n))})$
exponentielle	$\mathcal{O}(2^{poly(n)})$)
factorielle	$\mathcal{O}(n!)$

Table 1 – Lexique des différentes complexités d'un algorithme

Il existe d'autres complexités

Complexité en moyenne

- Il s'agit de la complexité en moyenne sur toutes les entrées ;
- Intéressant car certains problèmes ont une complexité dans le pire cas très élevées, mais les entrées qui correspondent à ces cas ne se produisent que très rarement en pratique ;
- Étudié essentiellement dans les algorithmes de tri (que nous verrons plus tard)

Complexité en mémoire

- On souhaite ici mesurer la mémoire utilisée par un algorithme ;
- Certains algorithmes sont en effet très rapide mais prennent beaucoup d'espace mémoire (ou inversement) ;

Langage de Description Algorithmique

Plus grand diviseur commun (pgcd)

Problème :

- Entrées : 2 entiers $a > b \geq 0$
- Sortie : le pgcd de a et b

Exemple : $pgcd(123, 82) = 41$

Plus grand diviseur commun (pgcd)

Problème :

- Entrées : 2 entiers $a > b \geq 0$
- Sortie : le pgcd de a et b

Exemple : $pgcd(123, 82) = 41$

Opérations élémentaires : $+, \times, -$ et division euclidienne

Plus grand diviseur commun (pgcd)

Problème :

- Entrées : 2 entiers $a > b \geq 0$
- Sortie : le pgcd de a et b

Exemple : $pgcd(123, 82) = 41$

Opérations élémentaires : $+, \times, -$ et division euclidienne

Observation : $\begin{cases} pgcd(a,b) & = pgcd(b,r) \quad \text{r reste de la div. de a par b} \\ pgcd(a,0) & = a \end{cases}$

Exemple : $pgcd(123, 82) = pgcd(82, 41) = pgcd(41, 0) = 41$

Pgcd : écriture d'un algorithme

Écriture en pseudo code

Observation : $\begin{cases} pgcd(a,b) & = pgcd(b,r) \quad \text{r reste de la div. de a par b} \\ pgcd(a,0) & = a \end{cases}$

Algorithme : ? ? ?

Pgcd : écriture d'un algorithme

Écriture en pseudo code

Observation : $\begin{cases} pgcd(a,b) & = pgcd(b,r) \quad \text{r reste de la div. de a par b} \\ pgcd(a,0) & = a \end{cases}$

Algorithme :

Pgcd : écriture d'un algorithme

Écriture en pseudo code

Observation : $\begin{cases} pgcd(a,b) & = pgcd(b,r) \quad \text{r reste de la div. de a par b} \\ pgcd(a,0) & = a \end{cases}$

Algorithme :

$x \leftarrow a ; y \leftarrow b$

$r \leftarrow$ reste de la div. euclidienne de x par y

Pgcd : écriture d'un algorithme

Écriture en pseudo code

Observation : $\begin{cases} pgcd(a,b) & = pgcd(b,r) \quad \text{r reste de la div. de a par b} \\ pgcd(a,0) & = a \end{cases}$

Algorithme :

$x \leftarrow a ; y \leftarrow b$

$r \leftarrow$ reste de la div. euclidienne de x par y
si $r = 0$ **alors retourner** y

Pgcd : écriture d'un algorithme

Écriture en pseudo code

Observation : $\begin{cases} pgcd(a,b) & = pgcd(b,r) \quad \text{r reste de la div. de a par b} \\ pgcd(a,0) & = a \end{cases}$

Algorithme :

$x \leftarrow a ; y \leftarrow b$

$r \leftarrow$ reste de la div. euclidienne de x par y
si $r = 0$ **alors retourner** y
sinon $x \leftarrow y$; $y \leftarrow r$
fin si

Pgcd : écriture d'un algorithme

Écriture en pseudo code

Observation : $\begin{cases} pgcd(a,b) & = pgcd(b,r) \quad \text{r reste de la div. de a par b} \\ pgcd(a,0) & = a \end{cases}$

Algorithme :

$x \leftarrow a ; y \leftarrow b$
répéter
    $r \leftarrow$ reste de la div. euclidienne de x par y
    si $r = 0$ **alors retourner** y
    sinon $x \leftarrow y$; $y \leftarrow r$
    fin si
fin répéter

Pgcd : écriture d'un algorithme

Écriture en pseudo code

Observation : $\begin{cases} pgcd(a,b) & = pgcd(b,r) \quad \text{r reste de la div. de a par b} \\ pgcd(a,0) & = a \end{cases}$

Algorithme :

pgcd(a,b) `/* a,b entiers,` $a > b \geq 0$ `*/`

$x \leftarrow a$; $y \leftarrow b$
répéter
  $r \leftarrow$ reste de la div. euclidienne de x par y
  si $r = 0$ **alors retourner** y
  sinon $x \leftarrow y$; $y \leftarrow r$
  fin si
fin répéter

Pgcd : écriture d'un algorithme

Écriture en pseudo code

Observation : $\begin{cases} pgcd(a,b) & = pgcd(b,r) \quad \text{r reste de la div. de a par b} \\ pgcd(a,0) & = a \end{cases}$

Algorithme :

pgcd(a,b) `/* a,b entiers,` $a > b \geq 0$ `*/`

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a$; $y \leftarrow b$

répéter

$r \leftarrow$ reste de la div. euclidienne de x par y

si $r = 0$ **alors retourner** y

sinon $x \leftarrow y$; $y \leftarrow r$

fin si

fin répéter

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

$r \leftarrow x \ \% \ y$

$x \leftarrow y$

$y \leftarrow r$

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y

Algorithme d'Euclide

pgcd(a,b)
    entier x, y, r
    si $b = 0$ **retourner** a **fin si**
    $x \leftarrow a; y \leftarrow b$
    tant que $y \neq 0$ **faire**
        $r \leftarrow x \ \% \ y$
        $x \leftarrow y$
        $y \leftarrow r$
    fin tant que
    retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

r ← x % y

$x \leftarrow y$

$y \leftarrow r$

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78		

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

$r \leftarrow x \ \% \ y$

$x \leftarrow y$

$y \leftarrow r$

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78	123456	

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

$r \leftarrow x \ \% \ y$

$x \leftarrow y$

y ← r

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78	123456	78

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

r ← x % y

$x \leftarrow y$

$y \leftarrow r$

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78	123456	78
60		

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

$r \leftarrow x \ \% \ y$

$x \leftarrow y$

$y \leftarrow r$

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78	123456	78
60	78	

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

$r \leftarrow x \ \% \ y$

$x \leftarrow y$

y ← r

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78	123456	78
60	78	60

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

r ← x % y

$x \leftarrow y$

$y \leftarrow r$

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78	123456	78
60	78	60
18		

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

$r \leftarrow x \ \% \ y$

$x \leftarrow y$

$y \leftarrow r$

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78	123456	78
60	78	60
18	60	

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

$r \leftarrow x \ \% \ y$

$x \leftarrow y$

$y \leftarrow r$

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78	123456	78
60	78	60
18	60	18

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

r ← x % y

$x \leftarrow y$

$y \leftarrow r$

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78	123456	78
60	78	60
18	60	18
6		

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

$r \leftarrow x \ \% \ y$

$x \leftarrow y$

$y \leftarrow r$

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78	123456	78
60	78	60
18	60	18
6	18	

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

$r \leftarrow x \ \% \ y$

$x \leftarrow y$

y ← r

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78	123456	78
60	78	60
18	60	18
6	18	6

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

r ← x % y

$x \leftarrow y$

$y \leftarrow r$

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78	123456	78
60	78	60
18	60	18
6	18	6
0		

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

$r \leftarrow x \ \% \ y$

$x \leftarrow y$

$y \leftarrow r$

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78	123456	78
60	78	60
18	60	18
6	18	6
0	6	

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

$r \leftarrow x \ \% \ y$

$x \leftarrow y$

y ← r

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78	123456	78
60	78	60
18	60	18
6	18	6
0	6	0

Algorithme d'Euclide

pgcd(a,b)
    entier x, y, r
    si $b = 0$ **retourner** a **fin si**
     $x \leftarrow a; y \leftarrow b$
    tant que $y \neq 0$ **faire**
        $r \leftarrow x \ \% \ y$
        $x \leftarrow y$
        $y \leftarrow r$
    fin tant que
    retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78	123456	78
60	78	60
18	60	18
6	18	6
0	6	0

Algorithme d'Euclide

pgcd(a,b)

entier x, y, r

si $b = 0$ **retourner** a **fin si**

$x \leftarrow a; y \leftarrow b$

tant que $y \neq 0$ **faire**

$r \leftarrow x \ \% \ y$

$x \leftarrow y$

$y \leftarrow r$

fin tant que

retourner(x)

pgcd(12345678,123456)

r	x	y
$\perp$	12345678	123456
78	123456	78
60	78	60
18	60	18
6	18	6
0	**6**	0

Algorithmes Récursifs

Complexité d'un algorithme

- La complexité permet d'évaluer l'efficacité d'un algorithme ;
- Elle permet de comparer des algorithmes indépendamment des machines ;
- On note $\mathcal{O}$ la complexité asymptotique (*dans le pire cas*).

Introduction

- Les **algorithmes récursifs** et les **fonctions récursives** sont fondamentaux en informatique. Un algorithme est dit récursif s'il s'appelle **lui-même** ;
- Les premiers langages de programmation qui ont introduit la récursivité sont `LISP` et `Algol 60` et maintenant tous les langages de programmation modernes proposent une implémentation de la récursivité ;
- On oppose généralement les algorithmes récursifs aux algorithmes dits impératifs ou itératifs qui s'exécutent sans invoquer ou appeler explicitement l'algorithme lui-même.

Lien avec les mathématiques : la factorielle

On souhaite calculer $n!$. On rappelle que pour tout entier $n \geq 0$

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times \ldots \times n$$

Lien avec les mathématiques : la factorielle

On souhaite calculer $n!$. On rappelle que pour tout entier $n \geq 0$

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times \ldots \times n$$

On peut déduire de cette définition la propriété importante suivante

$$\forall n \geq 1, n! = n \times (n-1)!$$

et donc si on sait calculer (n-1) ! alors on sait calculer n !.

Lien avec les mathématiques : la factorielle

On souhaite calculer $n!$. On rappelle que pour tout entier $n \geq 0$

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times \ldots \times n$$

On peut déduire de cette définition la propriété importante suivante

$$\forall n \geq 1, n! = n \times (n-1)!$$

et donc si on sait calculer (n-1) ! alors on sait calculer n !.

Par ailleurs, on sait que $0! = 1$. On sait donc calculer 1 !, puis 2 !, et par récurrence on peut établir qu'on sait calculer n ! pour tout entier $n \geq 0$.

Algorithme de la factorielle

L'algorithme récursif de calcul de la factorielle distingue deux cas. Le premier cas ne nécessite aucun calcul, le second utilise la fonction *fact* pour calculer $(n-1)!$

```
1: function fact(n)
2:     if n == 0 then
3:         return 1
4:     else
5:         return n × fact(n − 1)
```

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact(4) \rightarrow 4 \times fact(3)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact(4) \rightarrow 4 \times fact(3)$$

$$fact(4) \rightarrow 4 \times 3 \times fact(2)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact(4) \rightarrow 4 \times fact(3)$$

$$fact(4) \rightarrow 4 \times 3 \times fact(2)$$

$$fact(4) \rightarrow 4 \times 3 \times 2 \times fact(1)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact(4) \rightarrow 4 \times fact(3)$$

$$fact(4) \rightarrow 4 \times 3 \times fact(2)$$

$$fact(4) \rightarrow 4 \times 3 \times 2 \times fact(1)$$

$$fact(4) \rightarrow 4 \times 3 \times 2 \times 1 \times fact(0)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact(4) \rightarrow 4 \times fact(3)$$

$$fact(4) \rightarrow 4 \times 3 \times fact(2)$$

$$fact(4) \rightarrow 4 \times 3 \times 2 \times fact(1)$$

$$fact(4) \rightarrow 4 \times 3 \times 2 \times 1 \times fact(0)$$

$$fact(4) \rightarrow 4 \times 3 \times 2 \times 1 \times 1$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact(4) \rightarrow 4 \times fact(3)$$

$$fact(4) \rightarrow 4 \times 3 \times fact(2)$$

$$fact(4) \rightarrow 4 \times 3 \times 2 \times fact(1)$$

$$fact(4) \rightarrow 4 \times 3 \times 2 \times 1 \times fact(0)$$

$$fact(4) \rightarrow 4 \times 3 \times 2 \times 1 \times 1$$

$$fact(4) \rightarrow 24$$

Autre algorithme de factorielle

Considérons l'algorithme suivant :

```
1: function fact2(n)
2:     return n × fact(n − 1)
```

Autre algorithme de factorielle

Considérons l'algorithme suivant :

```
1: function fact2(n)
2:     return n × fact(n − 1)
```

Quel est le souci avec cette fonction ?

Autre algorithme de factorielle

Considérons l'algorithme suivant :

```
1: function fact2(n)
2:     return n × fact(n − 1)
```

Quel est le souci avec cette fonction ?

L'évaluation de $fact2(1)$ conduit à un calcul infini :

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact2(1) \rightarrow 1 \times fact2(0)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact2(1) \rightarrow 1 \times fact2(0)$$
$$fact2(1) \rightarrow 1 \times 0 \times fact2(-1)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact2(1) \rightarrow 1 \times fact2(0)$$
$$fact2(1) \rightarrow 1 \times 0 \times fact2(-1)$$
$$fact2(1) \rightarrow 1 \times 0 \times -1 \times fact2(-2)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact2(1) \rightarrow 1 \times fact2(0)$$

$$fact2(1) \rightarrow 1 \times 0 \times fact2(-1)$$

$$fact2(1) \rightarrow 1 \times 0 \times -1 \times fact2(-2)$$

$$fact2(1) \rightarrow \ldots$$

Règle implicite de conception d'un algorithme récursif

Règle implicite

Tout algorithme récursif doit distinguer plusieurs cas dont l'un au moins ne doit pas contenir d'appels récursifs.

Les cas non récursifs d'un algorithme récursif sont appelés *cas de bases*. Les conditions que doivent satisfaire les données dans ces cas de bases sont appelées *conditions de terminaison*.

Règle implicite de conception d'un algorithme récursif

Règle implicite

Tout algorithme récursif doit distinguer plusieurs cas dont l'un au moins ne doit pas contenir d'appels récursifs.

Les cas non récursifs d'un algorithme récursif sont appelés *cas de bases*. Les conditions que doivent satisfaire les données dans ces cas de bases sont appelées *conditions de terminaison*.

Est-ce-que c'est suffisant ?

Reprenons notre exemple de la factorielle

```
1: function fact3(n)
2:     if n == 0 then
3:         return 1
4:     else
5:         return n × fact3(n + 1)
```

Reprenons notre exemple de la factorielle

```
1: function fact3(n)
2:     if n == 0 then
3:         return 1
4:     else
5:         return n × fact3(n + 1)
```

Quel est le souci avec cette fonction ?

Reprenons notre exemple de la factorielle

```
1: function fact3(n)
2:     if n == 0 then
3:         return 1
4:     else
5:         return n × fact3(n + 1)
```

Quel est le souci avec cette fonction ?

L'évaluation de $fact3(1)$ conduit à un calcul infini :

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact3(1) \rightarrow 1 \times fact3(2)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact3(1) \rightarrow 1 \times fact3(2)$$

$$fact31 \rightarrow 1 \times 2 \times fact3(3)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact3(1) \rightarrow 1 \times fact3(2)$$

$$fact31 \rightarrow 1 \times 2 \times fact3(3)$$

$$fact31 \rightarrow 1 \times 2 \times 3 \times \ldots$$

Deuxième règle implicite

```
1: function fact3(n)
2:     if n == 0 then
3:         return 1
4:     else
5:         return n × fact3(n + 1)
```

Deuxième règle implicite

```
1: function fact3(n)
2:     if n == 0 then
3:         return 1
4:     else
5:         return n × fact3(n + 1)
```

Deuxième règle

À chaque appel récursif il faut se rapprocher des conditions de terminaison.

Petit exercice

Exercice

Écrire une fonction qui calcule la somme des inverses des carrés des n premiers entiers naturels non nuls.

Petit exercice

Exercice

Écrire une fonction qui calcule la somme des inverses des carrés des n premiers entiers naturels non nuls.

```
1: function sum_inv(n)
2:     if n == 1 then
3:         return 1
4:     else
5:         return 1/n^2 + sum_inv(n − 1)
```

Recherche dans un tableau

Algorithme de recherche d'un élément dans un tableau

```
1: function search(tab, e)
2:     i ← 0
3:     n ← tab.size
4:     while i < n do
5:         if tab[i] == e then
6:             return i
7:         i ← i + 1
8:     return nonTrouvé
```

Algorithme de recherche d'un élément dans un tableau

```
1: function search(tab, e)
2:     i ← 0
3:     n ← tab.size
4:     while i < n do
5:         if tab[i] == e then
6:             return i
7:         i ← i + 1
8:     return nonTrouvé
```

Quelle est la complexité de cette fonction ?

Algorithme de recherche d'un élément dans un tableau

```
1: function search(tab, e)
2:     i ← 0
3:     n ← tab.size
4:     while i < n do
5:         if tab[i] == e then
6:             return i
7:         i ← i + 1
8:     return nonTrouvé
```

Quelle est la complexité de cette fonction ? $\mathcal{O}(n)$

Recherche d'un élément dans un tableau

La complexité précédente est trop élevée, surtout sachant que la recherche dans un tableau est une opération de base utilisée dans de nombreux algorithmes.
Pour aller plus vite, on peut utiliser les **tableaux triés** et la **dichotomie**.

Exemple : La dichotomie

Definition

La recherche dichotomique, ou recherche par dichotomie (en anglais : *binary search*), est un algorithme de recherche pour trouver la position d'un élément dans un tableau trié.

Le principe

L'objectif est trouver la position d'un élément dans un tableau trié.

- On trouve l'élément m avec la position la plus centrale du tableau (si le tableau est vide on s'arrête) ;
- On compare la valeur de l'élément recherché avec l'élément m ;
- Si elle est plus petite, on recommence dans le sous-tableau de gauche, sinon dans le sous-tableau de droite.

L'algorithme de la dichotomie

```
1:  function dichotomie(tab, min, max, e)
2:      if min == max then
3:          if tab[min] = e then
4:              return min
5:          else
6:              return nonTrouvé
7:      mid ← (min + max)/2
8:      if tab[mid] < e then
9:          return dichotomie(tab, mid + 1, max, e)
10:     else
11:         return dichotomie(tab, min, mid, e)
```

L'algorithme de la dichotomie

```
1:  function dichotomie(tab, min, max, e)
2:      if min == max then
3:          if tab[min] = e then
4:              return min
5:          else
6:              return nonTrouvé
7:      mid ← (min + max)/2
8:      if tab[mid] < e then
9:          return dichotomie(tab, mid + 1, max, e)
10:     else
11:         return dichotomie(tab, min, mid, e)
```

Quelle est la complexité de cette fonction ?

L'algorithme de la dichotomie

```
1:  function dichotomie(tab, min, max, e)
2:      if min == max then
3:          if tab[min] = e then
4:              return min
5:          else
6:              return nonTrouvé
7:      mid ← (min + max)/2
8:      if tab[mid] < e then
9:          return dichotomie(tab, mid + 1, max, e)
10:     else
11:         return dichotomie(tab, min, mid, e)
```

Quelle est la complexité de cette fonction ? $\mathcal{O}(log_2(n))$

L'algorithme de la dichotomie itérative

```
1:  function dichotomie(tab, e)
2:      min ← 0
3:      max ← taille - 1
4:      while min < max do
5:          mid ← (min + max)/2
6:          if tab[mid] < e then
7:              min ← mid + 1
8:          else
9:              max ← mid
10:     if tab[min] == e then
11:         return min
12:     else
13:         return nonTrouvé
```

L'algorithme de la dichotomie itérative

```
1:  function dichotomie(tab, e)
2:      min ← 0
3:      max ← taille - 1
4:      while min < max do
5:          mid ← (min + max)/2
6:          if tab[mid] < e then
7:              min ← mid + 1
8:          else
9:              max ← mid
10:     if tab[min] == e then
11:         return min
12:     else
13:         return nonTrouvé
```

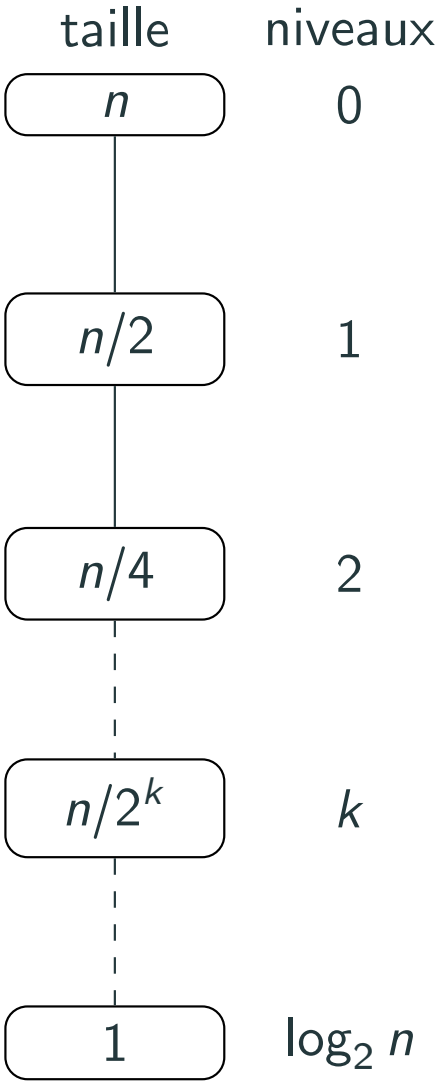
Quelle est la complexité de cette fonction ?

L'algorithme de la dichotomie itérative

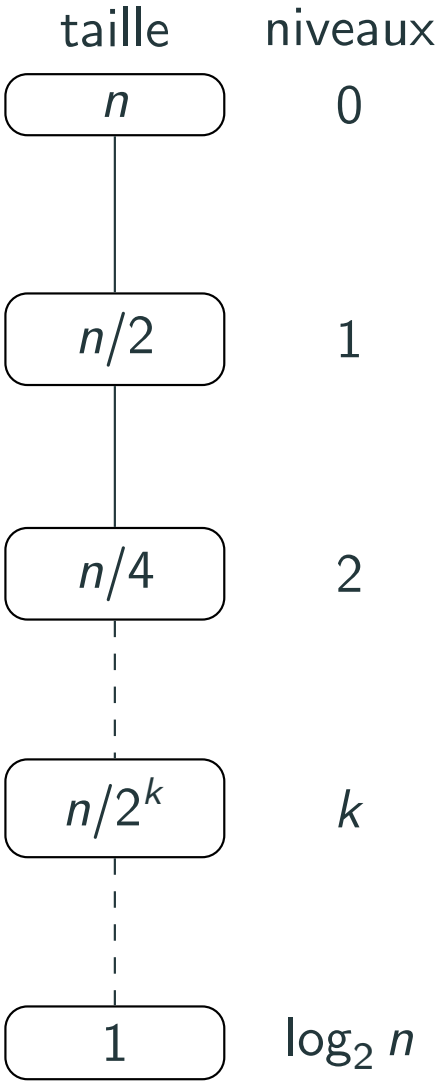
```
1:  function dichotomie(tab, e)
2:      min ← 0
3:      max ← taille - 1
4:      while min < max do
5:          mid ← (min + max)/2
6:          if tab[mid] < e then
7:              min ← mid + 1
8:          else
9:              max ← mid
10:     if tab[min] == e then
11:         return min
12:     else
13:         return nonTrouvé
```

Quelle est la complexité de cette fonction ? $\mathcal{O}(log_2(n))$

Complexité de la Recherche Dichotomique



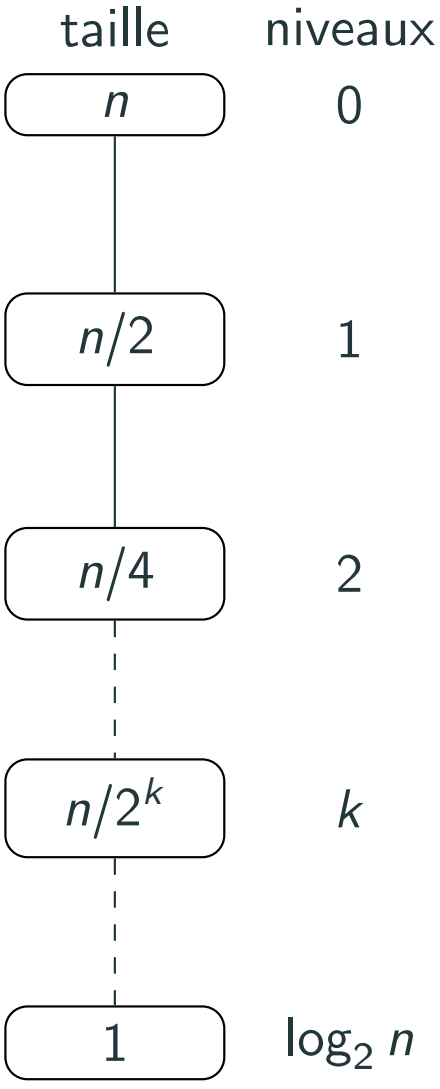
Complexité de la Recherche Dichotomique



Niveau k

- 1 sous problème de taille $n/2^k$
- $\mathcal{O}(1)$ opérations élémentaires par niveau

Complexité de la Recherche Dichotomique



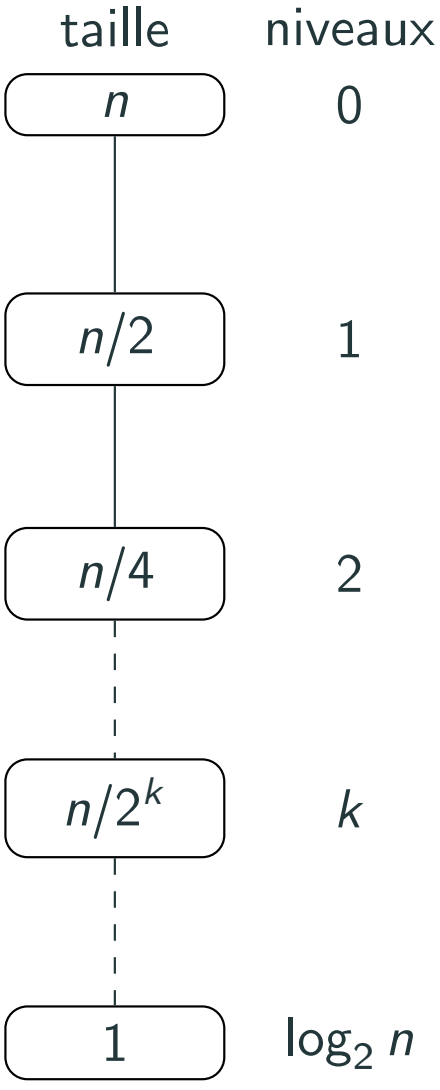
Niveau k

- 1 sous problème de taille $n/2^k$
- $\mathcal{O}(1)$ opérations élémentaires par niveau

Total :

- taille du problème divisée par 2 à chaque niveau $\Longrightarrow$ $\log_2 n + 1$ niveaux.

Complexité de la Recherche Dichotomique



Niveau k

- 1 sous problème de taille $n/2^k$
- $\mathcal{O}(1)$ opérations élémentaires par niveau

Total :

- taille du problème divisée par 2 à chaque niveau $\Longrightarrow$ $\log_2 n + 1$ niveaux.
- Complexité $= \mathcal{O}(\log n)$

Autres applications de la recherche dichotomique

- Jeu du nombre inconnu où l'on répond soit "plus grand" soit "plus petit" soit "gagné" ;
- Calcul d'une racine d'une fonction croissante ;
- Recherche de l'apparition d'un bug dans l'histoire d'un programme.

Tri de tableaux et algorithmes de tris

Insertion dans un tableau trié

```
1: function Insertion(tab, e)
2:     i ← taille
3:     while i > 0 and tab[i - 1] > e do
4:         tab[i] ← tab[i − 1]
5:         i ← i − 1
6:     tab[i] ← e
7:     taille ← taille + 1
```

Insertion dans un tableau trié

```
1: function Insertion(tab, e)
2:     i ← taille
3:     while i > 0 and tab[i - 1] > e do
4:         tab[i] ← tab[i − 1]
5:         i ← i − 1
6:     tab[i] ← e
7:     taille ← taille + 1
```

Quelle est la complexité de cette fonction ?

Insertion dans un tableau trié

```
1: function Insertion(tab, e)
2:     i ← taille
3:     while i > 0 and tab[i - 1] > e do
4:         tab[i] ← tab[i − 1]
5:         i ← i − 1
6:     tab[i] ← e
7:     taille ← taille + 1
```

Quelle est la complexité de cette fonction ? $\mathcal{O}(n)$

Tri par insertion

```
1:  function Insertsort(tab)
2:      i ← 1
3:      for i < taille do
4:          e ← tab[i]
5:          j ← i
6:          while j > 0 and tab[j - 1] > e do
7:              tab[j] ← tab[j − 1]
8:              j ← j − 1
9:          tab[j] ← e
10:         i ← i + 1
```

Tri par insertion

```
1:  function Insertsort(tab)
2:      i ← 1
3:      for i < taille do
4:          e ← tab[i]
5:          j ← i
6:          while j > 0 and tab[j - 1] > e do
7:              tab[j] ← tab[j − 1]
8:              j ← j − 1
9:          tab[j] ← e
10:         i ← i + 1
```

Quelle est la complexité de cette fonction ?

Tri par insertion

```
1:  function Insertsort(tab)
2:      i ← 1
3:      for i < taille do
4:          e ← tab[i]
5:          j ← i
6:          while j > 0 and tab[j - 1] > e do
7:              tab[j] ← tab[j − 1]
8:              j ← j − 1
9:          tab[j] ← e
10:         i ← i + 1
```

Quelle est la complexité de cette fonction ? $\mathcal{O}(n^2)$

Diviser pour régner

Approche diviser pour régner

En informatique, **diviser pour régner** est une technique algorithmique consistant à :

1. Diviser : découper un problème initial en sous-problèmes ;
2. Régner : résoudre les sous-problèmes (récursivement ou directement s'ils sont assez petits) ;
3. Combiner : calculer une solution au problème initial à partir des solution des sous-problèmes.

Approche diviser pour régner

En informatique, **diviser pour régner** est une technique algorithmique consistant à :

1. Diviser : découper un problème initial en sous-problèmes ;
2. Régner : résoudre les sous-problèmes (récursivement ou directement s'ils sont assez petits) ;
3. Combiner : calculer une solution au problème initial à partir des solution des sous-problèmes.

Comment obtenir un tableau trié, si l'on sait trier chaque moitié ?

Fusion de tableaux triés

```
1: function fusion(A[a_1, a_2, ..., a_n], B[b_1, b_2, ..., b_n])
2:     if A est vide then
3:         return B
4:     if B est vide then
5:         return A
6:     if A[a_1] <= B[b_1] then
7:         return A[a_1] + fusion(A[a_2, ..., a_n], B)
8:     else
9:         return B[b_1] + fusion(A, B[b_2, ..., b_n])
```

Fusion de tableaux triés

```
1: function fusion(A[a_1, a_2, ..., a_n], B[b_1, b_2, ..., b_n])
2:     if A est vide then
3:         return B
4:     if B est vide then
5:         return A
6:     if A[a_1] <= B[b_1] then
7:         return A[a_1] + fusion(A[a_2, ..., a_n], B)
8:     else
9:         return B[b_1] + fusion(A, B[b_2, ..., b_n])
```

Quelle est la complexité de cette fonction ?

Fusion de tableaux triés

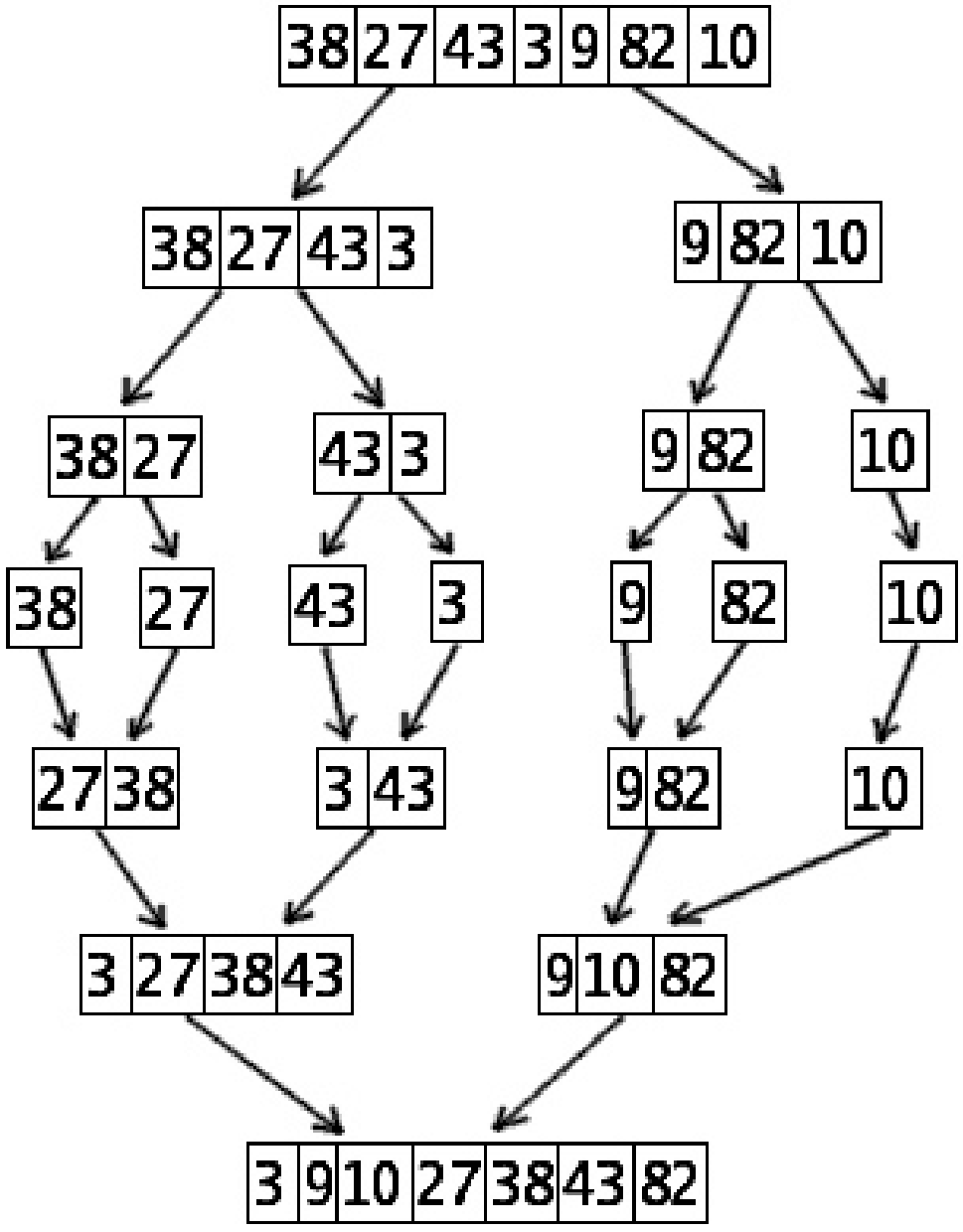
```
1: function fusion(A[a_1, a_2, ..., a_n], B[b_1, b_2, ..., b_n])
2:     if A est vide then
3:         return B
4:     if B est vide then
5:         return A
6:     if A[a_1] <= B[b_1] then
7:         return A[a_1] + fusion(A[a_2, ..., a_n], B)
8:     else
9:         return B[b_1] + fusion(A, B[b_2, ..., b_n])
```

Quelle est la complexité de cette fonction ? $\mathcal{O}(t_a + t_b)$

Tri par fusion (MergeSort)

```
1: function triFusion(T[t_1, t_2, ..., t_n])
2:     if t_n <= 1 then
3:         return T
4:     else
5:         return fusion(triFusion(T[t_1, ..., t_{n/2}]),
   triFusion(T[t_{n/2+1}, ..., t_n]))
```

Tri par fusion (MergeSort)



Tri par fusion (MergeSort)

```
1: function triFusion(T[t_1, t_2, ..., t_n])
2:     if t_n <= 1 then
3:         return T
4:     else
5:         return fusion(triFusion(T[t_1, ..., t_{n/2}]),
   triFusion(T[t_{n/2+1}, ..., t_n]))
```

Tri par fusion (MergeSort)

```
1: function triFusion(T[t_1, t_2, ..., t_n])
2:     if t_n <= 1 then
3:         return T
4:     else
5:         return fusion(triFusion(T[t_1, ..., t_{n/2}]),
   triFusion(T[t_{n/2+1}, ..., t_n]))
```

Quelle est la complexité de cette fonction ?

Tri par fusion (MergeSort)

```
1: function triFusion(T[t_1, t_2, ..., t_n])
2:     if t_n <= 1 then
3:         return T
4:     else
5:         return fusion(triFusion(T[t_1, ..., t_{n/2}]),
   triFusion(T[t_{n/2+1}, ..., t_n]))
```

Quelle est la complexité de cette fonction ? $\mathcal{O}(nlogn)$

Annexes : Complexité d'un algorithme diviser pour régner

La complexité

Le temps d'exécution d'un algorithme *diviser pour régner* se décompose suivant les trois étapes du paradigme de base.

- Diviser : le problème en **a** sous-problèmes chacun de taille **n/b**. Soit **D(n)** le temps nécessaire à la division du problème en sous-problèmes ;
- Régner : soit **a T(n/b)** le temps de résolution des **a** sous-problèmes ;
- Combiner : soit **C(n)** le temps nécessaire pour construire la solution finale à partir des solutions aux sous-problèmes.

Finalement, le temps d'exécution global de l'algorithme est :

$$T(n) = aT(n/b) + D(n) + C(n)$$

Soit la fonction $f(n)$ qui regroupe D(n) et C(n). T(n) est alors définie de la façon suivante :

$$T(n) = aT(n/b) + f(n)$$

La complexité

Le temps d'exécution d'un algorithme *diviser pour régner* se décompose suivant les trois étapes du paradigme de base.

- Diviser : le problème en **a** sous-problèmes chacun de taille **n/b**. Soit **D(n)** le temps nécessaire à la division du problème en sous-problèmes ;
- Régner : soit **a T(n/b)** le temps de résolution des **a** sous-problèmes ;
- Combiner : soit **C(n)** le temps nécessaire pour construire la solution finale à partir des solutions aux sous-problèmes.

Finalement, le temps d'exécution global de l'algorithme est :

$$T(n) = aT(n/b) + D(n) + C(n)$$

Soit la fonction $f(n)$ qui regroupe D(n) et C(n). T(n) est alors définie de la façon suivante :

$$T(n) = aT(n/b) + f(n)$$

On s'appelle ce théorème le *master theorem*

Théorème de résolution de la récurrence

Master theorem

$$T(n) = aT(n/b) + f(n)$$

Supposons que $f(n) = c * n^k$, on a : $T(n) = aT(n/b) + cn^k$

$$a > b^k \rightarrow T(n) = \mathcal{O}(n^{log_b a})$$

$$a = b^k \rightarrow T(n) = \mathcal{O}(n^k log_b n)$$

$$a < b^k \rightarrow T(n) = \mathcal{O}(f(n)) = O(n^k)$$

En utilisant le *master theorem*

- Diviser : le problème en **a** sous-problèmes chacun de taille **n/b**. Soit **D(n)** le temps nécessaire à la division du problème en sous-problèmes ;
- Régner : soit **a T(n/b)** le temps de résolution des **a** sous-problèmes ;
- Combiner : soit **C(n)** le temps nécessaire pour construire la solution finale à partir des solutions aux sous-problèmes.

```
1:  function dichotomie(tab, min, max, e)
2:      if min == max then
3:          if tab[min] == e then
4:              return min
5:      else
6:          return nonTrouvé
7:      mid ← (min + max)/2
8:      if tab[mid] < e then
9:          return dichotomie(tab, mid + 1, max, e)
10:     else
11:         return dichotomie(tab, min, mid, e)
```

Calcul de la complexité

Rappel

Supposons que $f(n) = c * n^k$, on a : $T(n) = aT(n/b) + cn^k$

$$a > b^k \rightarrow T(n) = \mathcal{O}(n^{log_b a})$$

$$a = b^k \rightarrow T(n) = \mathcal{O}(n^k log_b n)$$

$$a < b^k \rightarrow T(n) = \mathcal{O}(f(n)) = O(n^k)$$

D'après le théorème, on a donc :

- Pour la recherche dichotomique, a = 1, b = 2, k = 0 $\rightarrow$ a = b^k

$$T(n) = \mathcal{O}(n^k log_b n) = \mathcal{O}(log(n))$$

- Pour le tri fusion, a = 2, b = 2, k = 1 $\rightarrow$ a = b^k

$$T(n) = \mathcal{O}(n^k log_b n) = \mathcal{O}(nlog(n))$$