Rappel : ingrédients de base des algorithmes

Données

- Entrées
- Sorties
- Variables internes

Instructions

- Affectations
- Structures de contrôle
  - Branchements conditionnels (tests)
  - Itérations (boucles)
  - Boucles conditionnelles

Sous-algorithmes

Rappel: complexité temporelle et notation $\Theta(\cdot)$

Définition 1

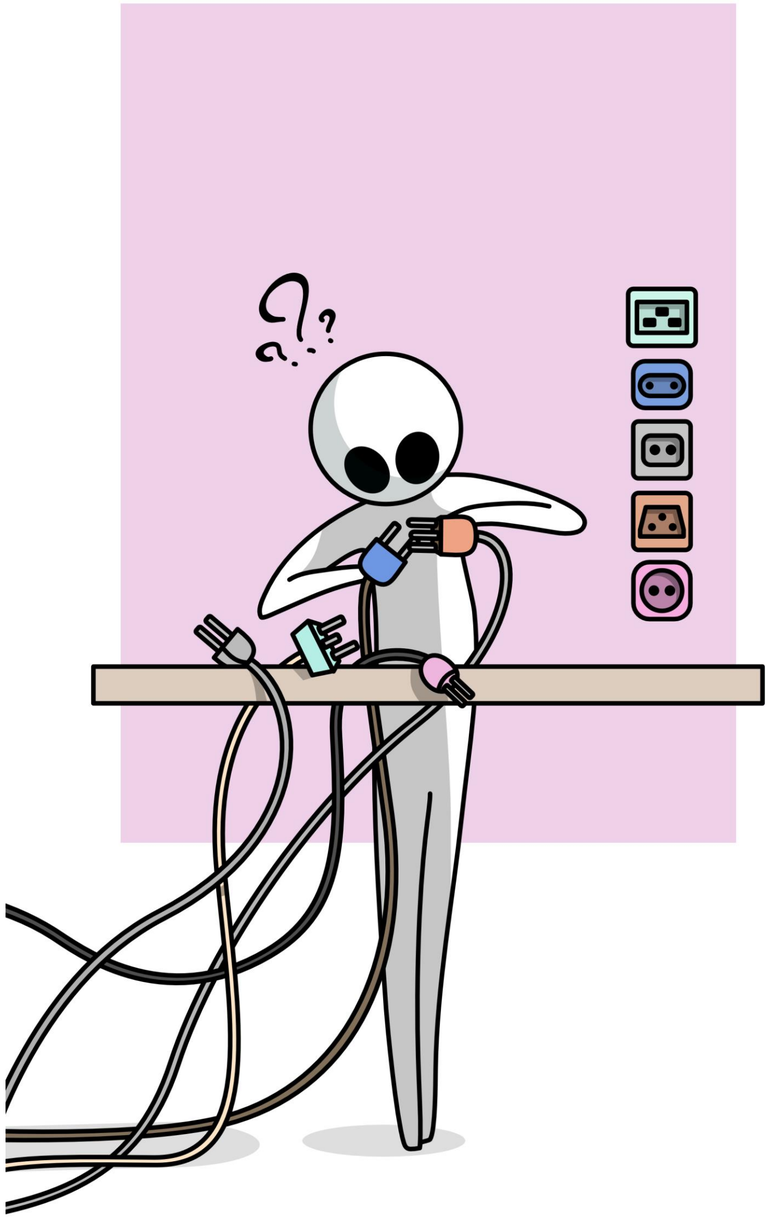
> La complexité temporelle d'un algorithme est le nombre **d'opérations élémentaires** effectuées au cours de son exécution, dans le **pire des cas**.

Définition 2

> Soient $f, g : \mathbb{N} \to \mathbb{R}_+$ deux fonctions non-négatives
> On dit que "$f(n)$ est un **grand theta** de $g(n)$" et on écrit "$f(n) = \Theta(g(n))$"
> s'il existe $0 < C_1 < C_2 < \infty$ et $N \geq 1$ tels que
>
> $$C_1\, g(n) \leq f(n) \leq C_2\, g(n) \qquad \text{pour tout } n \geq N$$

Deux exemples :

- Les fonctions $f(n) = n + 2$ et $f(n) = 3n + 3$ sont toutes deux des $\Theta(n)$
- La fonction $f(n) = \frac{n(n-1)}{2} + 1$ est un $\Theta(n^2)$



Information, Calcul et Communication

Complexité temporelle : un (autre) exemple concret

Olivier Lévêque

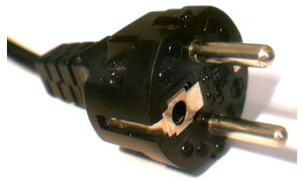
École polytechnique fédérale de Lausanne

Deux font la paire










Question : Parmi toutes les fiches et prises ci-dessus, y a-t-il une paire qui s'adapte l'une à l'autre?

Réécriture du problème avec des nombres entiers

En remplaçant les fiches et les prises par des nombres entiers positifs et négatifs, respectivement, la question précédente se transforme en :

- Etant donnée une liste L de n nombres entiers positifs et négatifs, existe-t-il $i, j \in \{1, \ldots, n\}$ tels que $i < j$ et $L(i) + L(j) = 0$?

Exemple : Si $L = (-15, -12, -3, -1, +5, +17, +23)$, alors la réponse est non.

Note : Vu que nous avons affaire ici à des nombres entiers, nous allons supposer de plus que la liste L en entrée est *ordonnée*.

Première méthode de résolution

Etant donnée une liste L de n nombres entiers positifs et négatifs, existe-t-il $i, j \in \{1, \ldots, n\}$ tels que $i < j$ et $L(i) + L(j) = 0$?

Deux font la paire

entrée : liste ordonnée L de nombres entiers
sortie : valeur binaire oui / non

$s \leftarrow non$
Pour i allant de 1 à $n - 1$:
    Pour j allant de $i + 1$ à n :
        Si $L(i) + L(j) = 0$, alors : $s \leftarrow oui$
Sortir : s

Complexité temporelle de cet algorithme: $\Theta(n^2)$

Les deux boucles imbriquées explorent toutes les paires possibles d'indices $i < j$ dans $\{1 \ldots n\}$, qui sont au nombre de

$$\underbrace{(n-1)}_{i=1} + \underbrace{(n-2)}_{i=2} + \cdots + 2 + 1 = \underbrace{\frac{n(n-1)}{2}}_{\text{nb de paires}} \quad \underbrace{\qquad\qquad}_{i=n-1}$$

donc la complexité temporelle de l'algorithme est $\Theta(n^2)$.

Complexité temporelle de cet algorithme: $\Theta(n^2)$

Les deux boucles imbriquées explorent toutes les paires possibles d'indices $i < j$ dans $\{1 \ldots n\}$, qui sont au nombre de

$$(n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2}$$

donc la complexité temporelle de l'algorithme est $\Theta(n^2)$.

Question : Peut-on faire mieux ?

Deuxième méthode de résolution

Deux font la paire

entrée : liste ordonnée L de nombres entiers
sortie : valeur binaire oui / non

Pour i allant de 1 à $n-1$:
    Pour j allant de $i+1$ à n :
        Si $L(i)+L(j)=0$, alors : Sortir : oui
Sortir : non

→ Complexité temporelle $\Theta(n^2)$ également : dans le pire des cas, l'algorithme doit parcourir toutes les paires (i,j) avant de sortir.

Remarque :

Aucun des deux algorithmes précédents n'exploite **l'ordre** de la liste L.

Autre idée : distinguer les nombres positifs et négatifs

- déterminer la valeur de $k \in \{1..n\}$ telle que $\begin{cases} L(i) < 0 & \text{si } i \leq k \\ L(i) > 0 & \text{si } i > k \end{cases}$ (on suppose que $L(i) \neq 0 \ \forall i$) — $\Theta(n)$

- Pour i allant de 1 à k
  Pour j allant de k+1 à n
  Si $L(i) + L(j) = 0$, sortir oui — $\frac{n^2}{4} = \Theta(n^2)$ ds le pire des cas ($k = n/2$)

- Sortir non

Troisième méthode de résolution

Deux font la paire

entrée : liste ordonnée L de nombres entiers
sortie : valeur binaire oui / non

$i \leftarrow 1$
$j \leftarrow n$
Tant que $i < j$:
    Si $L(i) + L(j) = 0$, alors : Sortir : oui
    Si $L(i) + L(j) < 0$, alors : $i \leftarrow i + 1$
    Si $L(i) + L(j) > 0$, alors : $j \leftarrow j - 1$
Sortir : non

→ Complexité temporelle de ce dernier algorithme: $\Theta(n)$ (une seule boucle !)

$n = 6$

$$L = (\cancel{-23}, \cancel{-18}, \cancel{-15}, +4, +12, \cancel{+27})$$

$i = 1, j = 6$

$i = 1, j = 5$

$i = 2, j = 5$

$i = 3, j = 5$

$i = 4, j = 5$

$-23 + 27 = +4 \quad \rightarrow \cancel{+27}$

$-23 + 12 = -11 \quad \rightarrow \cancel{-23}$

$-18 + 12 = -6 \quad \rightarrow \cancel{-18}$

$-15 + 12 = -3 \quad \rightarrow \cancel{-15}$

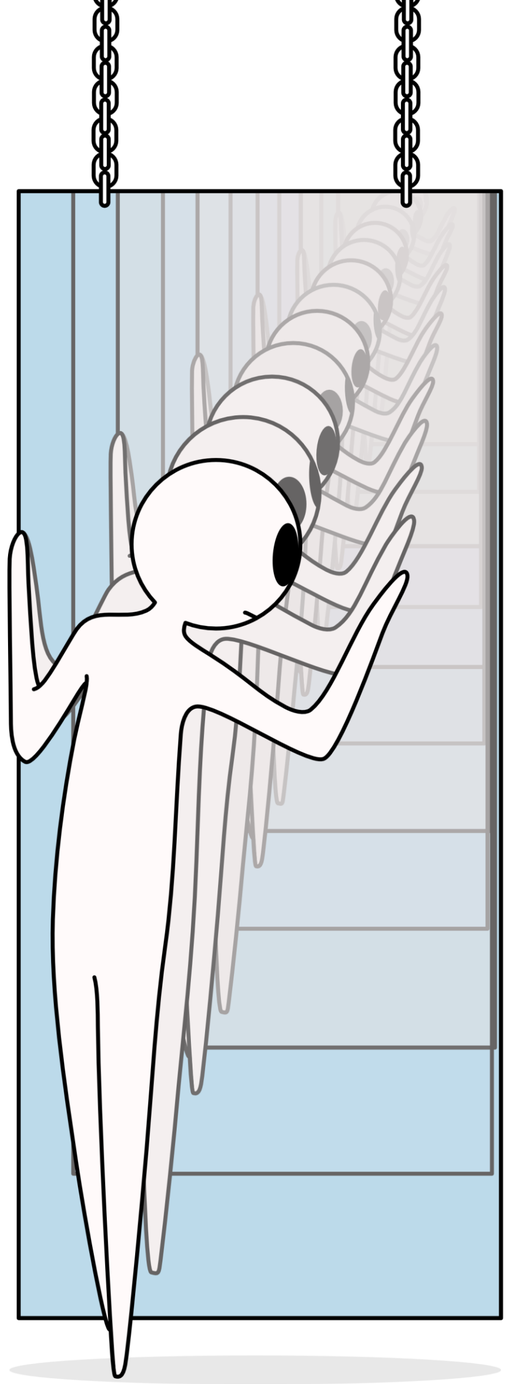
$+4 + 12 = +16 \quad \rightarrow \cancel{+12}$

$i = j = 4$ STOP

Conclusion

- Pour un problème donné, il existe souvent *plusieurs* algorithmes de résolution différents.

- En général, des données d'entrée *structurées* permettent une résolution *plus efficace* du problème.



Information, Calcul et Communication

Récursivité : introduction

École polytechnique fédérale de Lausanne

Olivier Lévêque

Définition

Un algorithme récursif est un algorithme qui résout un problème en calculant des solutions d'instances plus petites du même problème (l'algorithme s'invoquant lui-même de façon répétitive).

Exemple: recherche de clé

image: Freepik.com

Algorithme

recherche de clé (dans un carton donné)

est-ce que je vois la clé quelque part dans le carton?
- si oui, c'est bon: sortir de l'algorithme
- si non, est-ce que le carton contient au moins un autre carton ?
  - si oui, parcourir la liste des autres cartons et effectuer une **recherche de clé** dans chacun d'entre eux
  - si non, sortir de l'algorithme

Deux remarques

- cet algorithme se termine toujours (mais pas forcément avec succès, si la clé n'est pas dans le carton donné)
- remplacer "carton" par "camionnette" pour lancer l'algorithme au départ

La récursivité: trois principes

1. Un algorithme récursif a toujours une **condition de terminaison** (correspondant à l'instance la plus simple du problème à résoudre).

2. Pour résoudre un problème donné, un algorithme récursif fait d'abord appel à lui-même avec des données de taille plus petite en entrée (résolvant ainsi une **instance plus simple** du problème).

3. Un **processus de reconstruction** est généralement nécessaire ensuite pour obtenir la solution du problème d'origine.

Illustration

Calcul de la somme des n premiers nombres entiers

Exemple: lorsque $n = 3$, $s(3) = 1 + 2 + 3 = 6$

somme (version itérative)

entrée : nombre entier positif n
sortie : $s(n) =$ somme des n premiers nombres entiers

$s \leftarrow 0$
Pour i allant de 1 à n :
  $s \leftarrow s + i$
Sortir : s

Complexité temporelle: $\Theta(n)$ (une boucle)

Version récursive

instance plus simple du problème

- Résolution incrémentale : $s(n) = n + s(n-1)$

recombinaison

- Condition de terminaison : $n = 1$ (sortie correspondante : $s(1) = 1$)

Version récursive

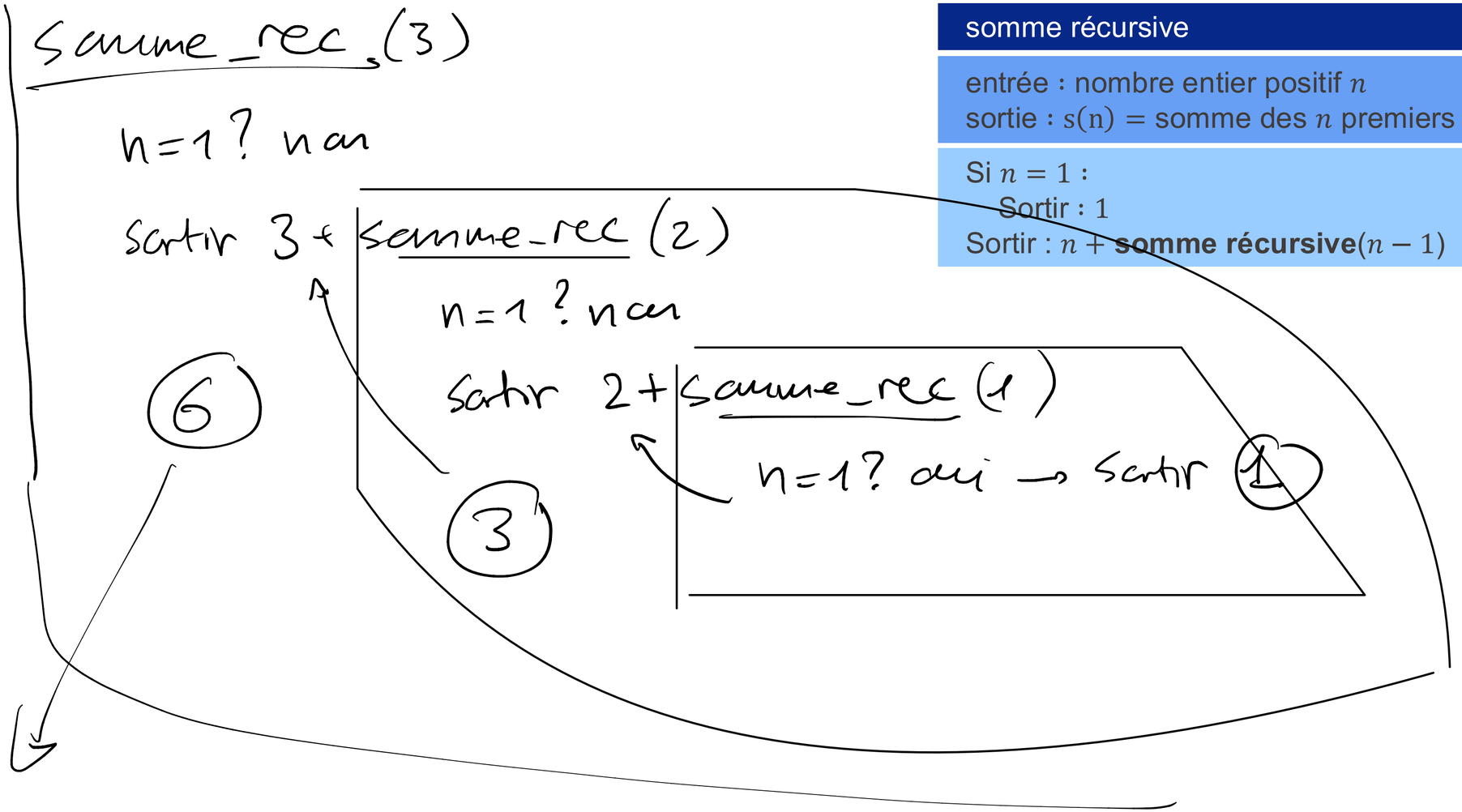
somme récursive

entrée : nombre entier positif n
sortie : $\mathrm{s(n)}$ = somme des n premiers nombres entiers

Si $n = 1$:
    Sortir : 1
Sortir : n + **somme récursive**$(n-1)$

Complexité temporelle: également $\Theta(n)$ (comme nous allons le voir)

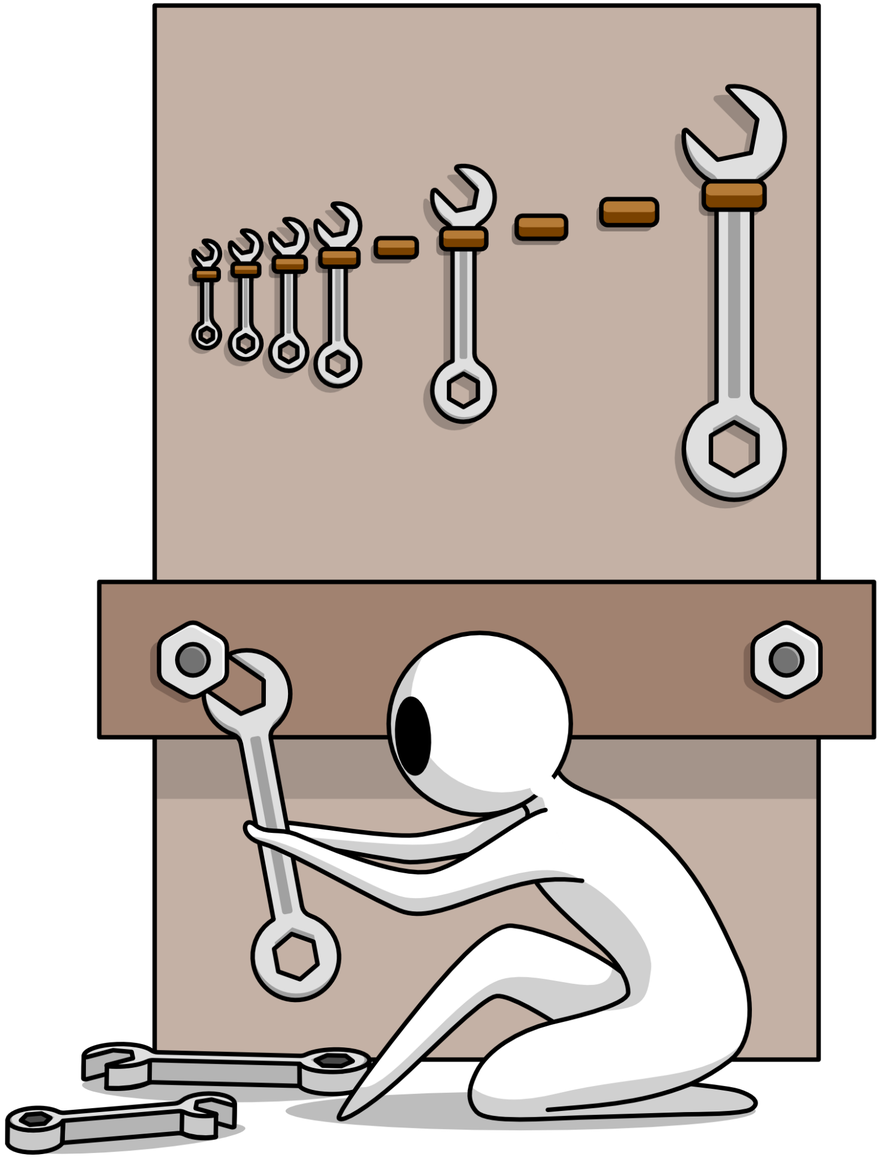
Schéma d'exécution pour $n = 3$



somme récursive

entrée : nombre entier positif n
sortie : $s(n)$ = somme des n premiers nombres entiers

Si $n = 1$:
  Sortir : 1
Sortir : n + **somme récursive**$(n - 1)$



Information, Calcul et Communication

Recherche par dichotomie

École polytechnique fédérale de Lausanne

Olivier Lévêque

Recherche d'un élément dans une liste

Problème

Identifier si un élément fait partie d'une liste donnée est une composante essentielle de nombreux algorithmes. On distingue deux cas principaux :

1. **Recherche dans une liste non-ordonnée**
   - Est-ce qu'un ingrédient donné (ex: gluten) fait partie ou non de la liste des ingrédients d'un produit ?
   - Est-ce que la feuille de papier sur laquelle j'avais écrit ce numéro de téléphone se trouve dans cette pile de feuilles que j'ai devant moi ?

Dans ce cas, pour identifier si l'élément qu'on recherche fait partie de la liste ou non, on n'a pas d'autre choix que de parcourir toute la liste.

Recherche linéaire

recherche linéaire

entrée : liste L d'objets, de taille n, objet x
sortie : est-ce que $x \in L$? (oui/non)

Pour i allant de 1 à n :
    Si $L(i) = x$, alors : Sortir : oui
Sortir : non

La complexité temporelle de l'algorithme ci-dessus est $\Theta(n)$.

(Dans le pire des cas, i.e., lorsque $x \notin L$, il faut parcourir toute la liste.)

Recherche d'un élément dans une liste (bis)

2. Recherche dans une liste ordonnée

- Identification d'un numéro de carte de crédit
- Recherche d'un nom dans un annuaire (ordre lexicographique)

Dans ce cas, on peut bien sûr effectuer la même recherche linéaire que précédemment, mais on peut aussi faire beaucoup mieux grâce à la récursivité. C'est l'algorithme de **recherche par dichotomie**.

$L = (-13, -7, -4, +18, +255, +310, +500)$

$x = +10$

(premier essai)

? dichotomie (L, n)

$m \leftarrow \lfloor \frac{n}{2} \rfloor$ (partie entière inférieure de $\frac{n}{2}$)

ex: si $n = 11$, $\lfloor \frac{n}{2} \rfloor = 5$

si $x \leq L(m)$,

chercher x dans la partie gauche de L

$L(1:m)$

sinon

chercher x dans la partie droite de L

$L(m+1:n)$

Recherche dichotomique

dichotomie

entrée : liste ordonée L de nombres entiers, de taille n, objet x
sortie : est-ce que $x \in L$? (oui/non)

Cond. term.

Si $n = 1$, alors : si $x = L(1)$, alors : Sortir oui
sinon : Sortir non

$m \leftarrow \lfloor n/2 \rfloor$
Si $x \leq L(m)$, alors : Sortir **dichotomie**$(L(1:m), m, x)$
Sinon : Sortir **dichotomie**$(L(m+1:n), n-m, x)$

Schéma d'exécution de l'algorithme

Avec $L = (1,7,9,14)$, $n = 4$ et $x = 10$:

dicho (L, n, x)

n=1 ? non

m ← 2

x ≤ L(2) ? non

(10 ≤ 7 ?) ↳ sortir

dicho ((9,14), 2, 10)

n=1 ? non

m ← 1

10 ≤ 9 ? non → sortir

dicho ((14), 1, 10)

n=1 ? oui

→ 10=14 ? non

→ Sortir (non)

dichotomie

entrée : liste ordonée L de nombres entiers,
de taille n, objet x
sortie : est-ce que $x \in L$? (oui/non)

Si $n = 1$, alors : si $x = L(1)$, alors : Sortir oui
sinon : Sortir non
$m \leftarrow \lfloor n/2 \rfloor$
Si $x \leq L(m)$, alors : Sortir **dichotomie**$(L(1:m), m, x)$
Sinon : Sortir **dichotomie**$(L(m+1:n), n-m, x)$

Complexité temporelle de l'algorithme

- A chaque étape, la taille de la liste est divisée (approx.) par 2.
- Partant d'une liste de taille n, le nombre d'étapes nécessaires pour arriver à une liste de taille 1 est donc (approx.) $\log_2(n)$.
- → complexité temporelle $\Theta(\log_2(n))$:
  bien plus efficace que l'algorithme de recherche linéaire vu avant !

$\frac{n}{2^x} = 1$

Def: $\log_2(n) = x$ tel que $2^x = n$

$=$ "nombre de fois qu'il faut diviser n par 2 pour arriver à 1"

n	$\log_2 n$
1000	~10
1 mio	~20
1 mia	~30