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**.

d'instructions (pour ce cours)

= approx.

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 $\Theta(n^2)$

Autre ex :

Etant donné L une liste de n nombres entiers relatifs triés dans l'ordre croissant, identifier si oui ou non, il existe $i<j$ (avec $i, j \in \{1 \dots n\}$) tels que $L(i) + L(j) = 0$.

Ex : $L = (-17, -12, -5, -3, +4, +22)$

$\longrightarrow$ réponse : non

Algo 1

entrée : liste L, de taille n

sortie : oui / non

Pour i allant de 1 à n-1
| Pour j allant de i+n à n
| | Si L(i) + L(j) = 0, sortir oui

sortir non

pire des cas : $\nexists\ i, j$ tq $L(i) + L(j) = 0$

comp. temp $= \Theta(n^2)$

Algo 2 (L, n)

$\Theta(n)$ {
- $k = 1$
- Tant que $L[k] < 0$:
  - $k \leftarrow k+1$

}

$\Theta(n^2)$ {
- Pour i allant de 1 à k-1
  - Pour j allant de k à n
    - Si $L(i) + L(j) = 0$, sortir ceci

- Sortir rien

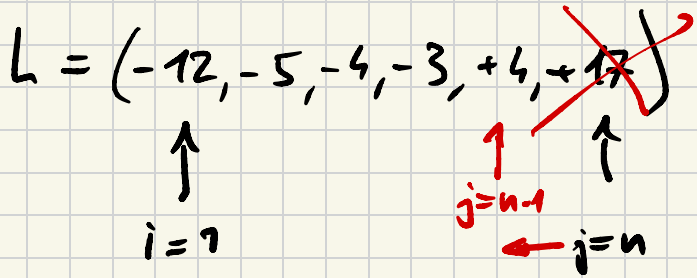
}

$\Theta(n) + \Theta(n^2) = \Theta(n^2)$

pire des cas:

Contient de nbs positifs que de nbs négatifs

Comp. temp : $\Theta(n^2)$



Algo 3 (L, n)

$i \leftarrow 1$

$j \leftarrow n$

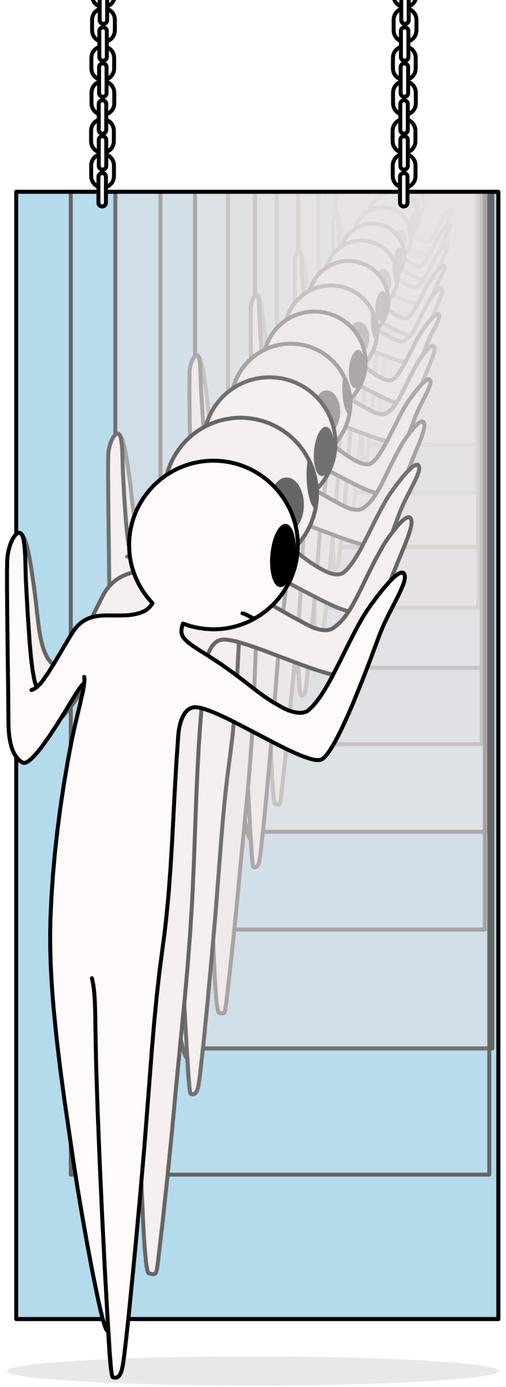
Tant que $i < j$:

| si $L(i)+L(j) > 0$, alors $j \leftarrow j-1$
| si $L(i)+L(j) < 0$, alors $i \leftarrow i+1$
| si $L(i)+L(j) = 0$, alors sortir oui

Sortir non

pire des cas : $\nexists\, i,j$ tq $L(i)+L(j)=0$

comp temp $= \Theta(n)$



Information, Calcul et Communication

Récursivité : introduction

Olivier Lévêque

École polytechnique fédérale de Lausanne

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)

$$\log_b(n) = \frac{\log_a(n)}{\log_a(b)} = c \cdot \log_a(n)$$

[en vrai : $\Theta(n \log_2 n)$]

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 : $s(n)$ = somme des n premiers nombres entiers

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

cond. de terminaison

récombinaison

instance plus simple du pb.

Version récursive

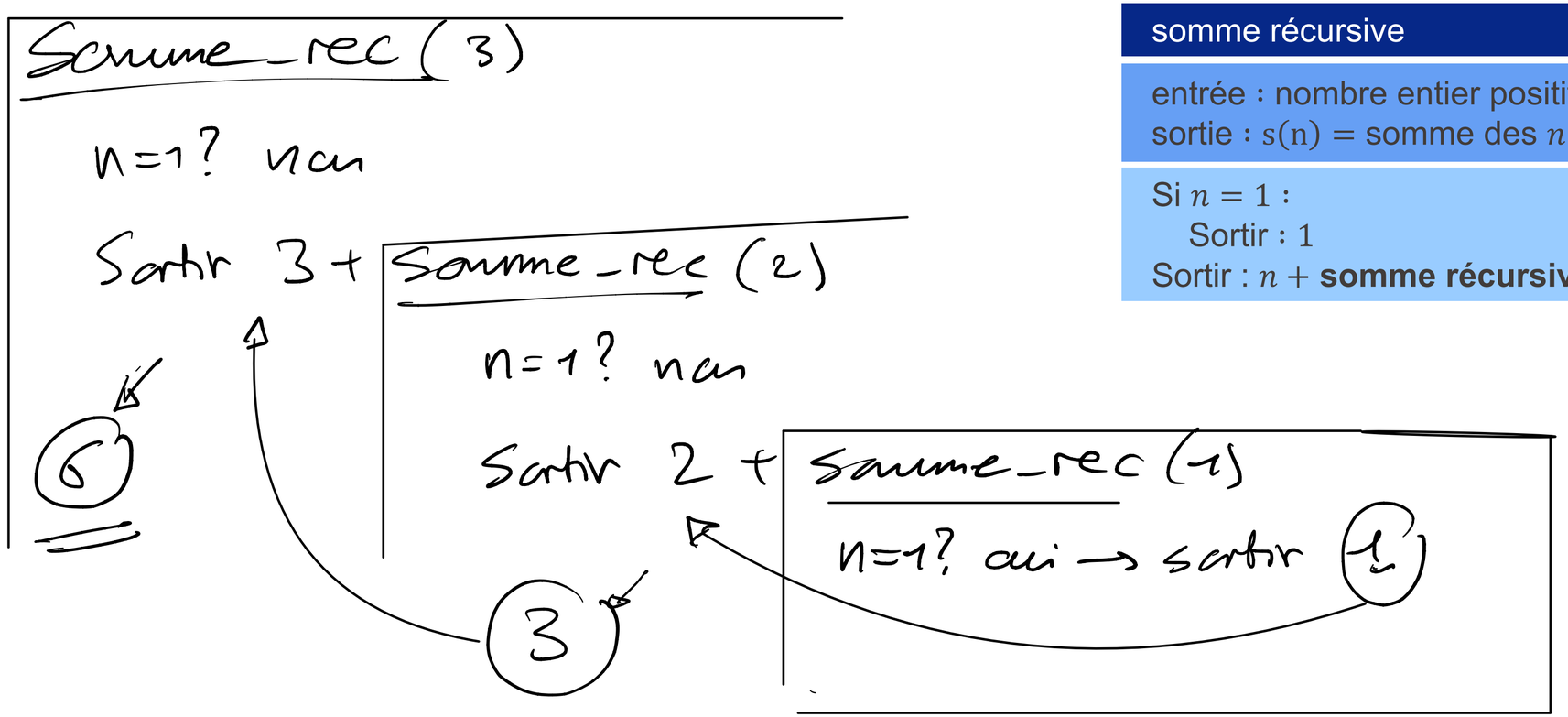
somme récursive

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

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

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

Schéma d'exécution pour $n = 3$

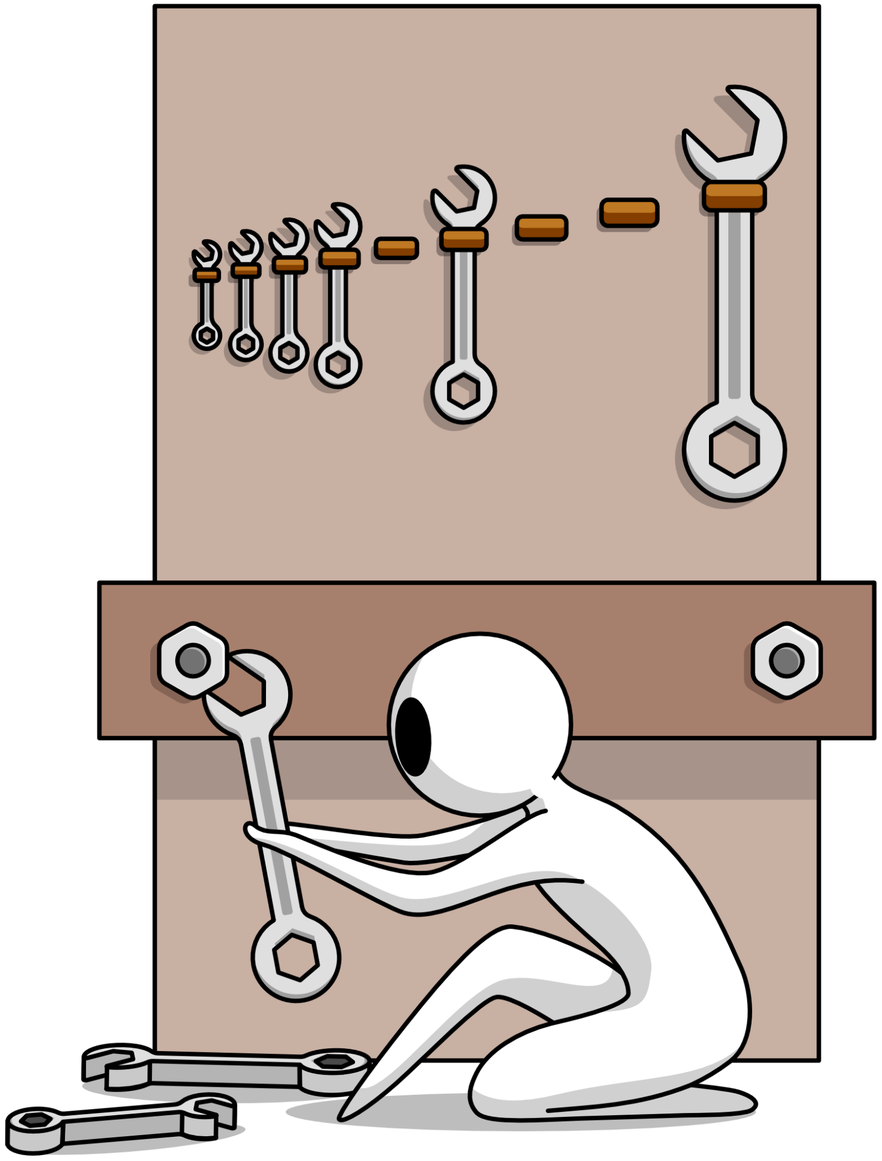


somme récursive

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

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

comp. temp. $\simeq$ nb de boîtes ouvertes $= \Theta(n)$



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**.

Recherche dichotomique

dichotomie

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

$x = -3$, $n = 8$

$L = (-17, -16, -5, -4, +3, +10, +15, +17)$

$x > L(4)$

Recherche dichotomique

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

} ← condition de terminaison

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

} réduction du pb

$m \leftarrow \lfloor n/2 \rfloor$ = partie entière de $n/2$

$L(1:m) = (L(1), L(2), \ldots, L(m))$ — m éléments

$n = 15 \rightarrow \frac{n}{2} = 7{,}5$, $\lfloor \frac{n}{2} \rfloor = 7$

$L(m+1:n) = (L(m+1), L(m+2) \ldots L(n))$ — $n-m$ éléments

Schéma d'exécution de l'algorithme

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

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**$(L(m+1:n), n-m, x)$

dicho (L, 4, 10)

n=1? non

m ← 2

10 ≤ 7 ? non → sortir

dicho (L'=(9,14), 2, 10)

n=1? non

m ← 1

10 ≤ 9 ? non → sortir

dicho (L''=(14), 1, 10)

n=1? oui

→ 10=14 ? non

→ sortir (non)

pire des cas : $x \notin L$

NB : pas besoin de recombiner ici !

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 !

Def: $x = \log_2(n) \iff 2^x = n \iff \frac{n}{2^x} = 1$

$x \simeq$ nombre de fois qu'il faut diviser n par 2 pour arriver à 1