UCLouvain

# Algorithmique

Cours préparatoires au master en sciences informatiques

Vincent Branders

2022–2023

# Notions de meilleurs et pires cas

# Un cas concret



**Algorithm:** firstNegative
**Input:** $A$, un tableau de $n$ entiers comprenant au moins un entier négatif
**Output:** Renvoie le premier entier negatif de $A$

```
for i ← 0 to n − 1 do
    if A[i] < 0 then
        return A[i]
```

**Algorithm:** firstNegativeRev
**Input:** $A$, un tableau de $n$ entiers comprenant au moins un entier négatif
**Output:** Renvoie le premier entier negatif de $A$

```
for i ← n − 1 down to 0 do
    if A[i] < 0 then
        negative ← A[i]
return negative
```

- Déterminez le temps de calcul[1] pour les instances suivantes :
  - $[-1]; [-7]; [-5, 1, 2]; [2, 4, -3, 6]; [3, -2, 4, 6, -3, -7]$
- Supposons une instance de taille $n = 15$, de quoi dépend le temps de calcul ?
  - Proposez une instance "particulièrement simple" à résoudre et une autre "particulièrement difficile"

Et avec `firstNegativeRev` ?

[1] Considérons ici simplement le nombre de branchements "if"

# L'importance des instances



- Pour caractériser l'efficacité des algorithmes, on ne peut se passer des instances à résoudre
- ...mais :
  - Il y a souvent un nombre infini d'instances possibles
  - Selon l'instance particulière considérée, un algorithme peut prendre plus ou moins de temps
  - Les instances possibles peuvent alors être classées en meilleur(s) cas, pire(s) cas ou cas moyen(s)
    - Attention, on parle bien d'instances de même taille
      ex. : si l'on s'intéresse au tri, y a t'il des instances qui prennent plus de temps à résoudre que d'autres si la taille du tableau est inchangée ?
  - Nous nous intéressons souvent au temps pris dans le pire cas car
    - nous voulons une garantie sur le temps maximum d'exécution
    - le meilleur cas donne lieu à une estimation optimiste
    - un cas représentatif "moyen" est souvent difficile à définir

# Trouver le pire cas : `selectionDecSort`



**Algorithm:** selectionDecSort
**Input:** Un tableau $A$ de $n$ éléments
**Output:** Le tableau $A$ est trié en ordre décroissant

**for** $i \leftarrow 0$ **to** $n-1$ **do**
  $pos \leftarrow i$
  **for** $j \leftarrow i+1$ **to** $n-1$ **do**
    **if** $A[j] > A[pos]$ **then**
      $pos \leftarrow j$
  **if** $A[i] < A[pos]$ **then**
    `swap`$(A[i], A[pos])$
**return** $A$

- Identifiez le(s) pire(s) cas en terme de nombre d'échanges (`swap`)
  1. $A = [\ 8,\ 7,\ 6,\ 5,\ 4,\ 3] \rightarrow 0$
  2. $A = [\ 2, -7,\ 0,\ 4,\ 9,\ 11] \rightarrow 5 \rightarrow \checkmark$
  3. $A = [\ 0,\ 0,\ 0,\ 0,\ 0,\ 0] \rightarrow 0$
  4. $A = [17,\ 17,\ 17,\ 15,\ 15,\ 15] \rightarrow 0$
  5. $A = [\ 2,\ 3,\ 4,\ 5,\ 6,\ 7] \rightarrow 3$

# Trouver le pire cas : `surprise`



**Algorithm:** surprise
**Input:** Un entier strictement positif $n$
**Output:** Un tableau $A$ de taille $n+1$ vérifiant une propriété à découvrir

```
for i ← 0 to n do
    temp ← i
    for j ← 1 to 3 do
        temp ← temp × i
    A[i] ← temp
return A
```

- Identifiez le(s) pire(s) cas en terme de nombre d'opérations effectuées
  1. $n = 3 \quad \Rightarrow \times$
  2. $n = 1 \quad \Rightarrow \times$
  3. $n = 1000 \quad \Rightarrow \times$
  4. $n = 7 \quad \Rightarrow \times$
  5. $n = \infty \quad \Rightarrow \times$
  6. $n = 9 \quad \Rightarrow \times$

Il faut bien comparer pour des tailles identiques

# Identifier le(s) pire(s) cas



- Il faut considérer des problèmes d'une taille fixe pour distinguer les pires cas, meilleurs cas et cas moyens
- Ce sont les valeurs spécifiques qui définissent une instance particulière qui doivent être considérées
- Chercher les "limites" de (la portée de) l'algorithme
  - penser à des exemples de petites tailles d'abord
  - penser à des cas d'égalité dans la décision ou l'ordre des opérations (ex. trouver "le" premier entier négatif)
  - penser à des cas limites (ex. très grands, très petits, collection vide, collection déjà triée, collection triée en sens inverse, uniquement des valeurs négatives, ...)

# Analyse asymptotique



## Objectif

- Analyser le temps (ou l'espace) requis, en se concentrant sur l'algorithme et l'influence de la taille du problème, généralement dans le pire cas

- Complexité temporelle : analyse asymptotique du nombre d'opérations effectuées
- Complexité spatiale : analyse asymptotique de l'espace utilisé

## Propriété

L'analyse asymptotique s'intéresse à l'évolution de la complexité lorsque la taille du problème augmente ($\lim_{n \to \infty}$)

Par exemple, si la taille $n$ du problème est multipliée par 10 comment évolue le temps $T = f(n)$ ?

Si $f(n) = c \Rightarrow f(10n) = c$ — $T$ est inchangé

Si $f(n) = c.n \Rightarrow f(10n) = c.(10n) = 10f(n)$ — $T \times 10$

Si $f(n) = c.n^2 \Rightarrow f(10n) = c.(10n)^2 = 100f(n)$ — $T \times 100$

- La vitesse du processeur est un des facteurs qui conditionnent la valeur de la constante $c$
- La vitesse du processeur ne change rien au rapport $\frac{f(10n)}{f(n)}$

**En résumé**

- Une constante est tout ce qui ne dépend pas de la taille du problème, même si elle *peut* varier ! (processeur, langage, compilateur, ...)
- Si l'on s'intéresse à l'influence de la taille du problème sur le temps calcul, on peut donc négliger les constantes ⚠⚠⚠

# Opérations primitives



**Définition**

Une opération primitive

- est une instruction en langage de haut niveau (par exemple Python ou une description en pseudo-code)
- représente un nombre constant d'opérations élémentaires effectivement exécutées sur le processeur, une fois le programme compilé ou interprété

# Opérations primitives : exemples



- une affectation d'une valeur à une variable : $x = 10$
- une comparaison de deux nombres : $x < y$
- un branchement : *if ... then ... else ...*
- une opération arithmétique élémentaire : $i + 2$
- un accès à un élément d'un tableau : $A[i]$
- une instruction *return* dans une méthode
- une instruction d'appel à une méthode : $p.maMethode()$
  ($\neq$ l'éxécution de l'ensemble de $maMethode()$ !)

## En résumé

Comme on néglige les constantes, il suffit de "compter" les opérations primitives plutôt que les opérations élémentaires

Pourquoi se soucier du temps d'exécution ?

# Pourquoi se soucier du temps d'exécution ?



## Hypothèse

On peut traiter une opération primitive en 1 nanoseconde ($1.10^{-9}s$)

- $f(n)$ désigne le nombre d'opérations primitives effectuées en fonction de la taille $n$ du problème
- Combien de temps prend le programme pour terminer son exécution si $n = 1000$ selon $f(n)$ ?

| $f(n)$ | Temps |
|---|---|
| $n$ | $0.000\ 001$ s |
| $400n$ | $0.000\ 4$ s |
| $2n^2$ | $0.002$ s |
| $n^5$ | $\sim 11.5$ jours |
| $2^n$ | $3.4 \times 10^{284}$ années |

On considère que l'Univers existe depuis $13.8 \times 10^9$ années

# Quelle est la taille maximale que l'on peut traiter ?



- On cherche $n$ tel que le nombre d'opérations primitives effectuées pour un problème de taille $n$ est égal au nombre d'opérations primitives qu'on sait effectuer en un temps $t$.
- En $1$ seconde, on sait effectuer $10^9$ opérations primitives

  $\implies f(n) = \frac{t}{10^{-9}\text{s}}$
- Si $f(n) = n$ et $t = 1$s, $n = \frac{1}{10^{-9}}$, on peut résoudre un problème de taille $n = 10^9$ en $1$ seconde
- Si $f(n) = 2n^2$ et $t = 60$s :

  $2n^2 = \frac{60}{10^{-9}} \implies n^2 = \frac{60\times 10^9}{2} \implies n = \sqrt{3 \times 10^{10}} \approx 173,205$

| $f(n)$ | en 1 seconde | en 1 minute | en 1 heure |
|---|---:|---:|---:|
| $n$ | $1 \times 10^9$ | $6 \times 10^{10}$ | $3.6 \times 10^{12}$ |
| $400n$ | $2.5 \times 10^6$ | $1.5 \times 10^8$ | $9 \times 10^9$ |
| $2n^2$ | $22,360$ | $173,205$ | $1.3 \times 10^6$ |
| $n^5$ | $63$ | $143$ | $324$ |
| $2^n$ | $29$ | $35$ | $41$ |

# Quelle est la taille maximale que l'on peut traiter ?

*(suite)*



Si $m$ est la taille maximale que l'on pouvait traiter en un temps donné, que devient $m$ si l'on acquièrt un processeur **256** fois plus rapide ?

- Pour $f(n) = 400n$, on avait $400n = \frac{t}{10^{-9}} \implies n = \frac{t}{400 \times 10^{-9}}$ donc $m = \frac{t}{400 \times 10^{-9}}$
  maintenant, $400n = \frac{t}{\frac{10^{-9}}{256}} = \frac{256.t}{10^{-9}} \implies n = \frac{256.t}{400 \times 10^{-9}}$ et donc $n = 256m$
- Pour $f(n) = n^5$, on avait $n^5 = \frac{t}{10^{-9}} \implies n = \sqrt[5]{\frac{t}{10^{-9}}}$ donc $m = \sqrt[5]{\frac{t}{10^{-9}}}$
  maintenant, $n^5 = \frac{256.t}{10^{-9}} \implies n = \sqrt[5]{\frac{256.t}{10^{-9}}}$ et donc $n = \sqrt[5]{256} \times m = 3.03m$

| $f(n)$ | Nouvelle taille maximale |
|---|---|
| $n$ | $256m$ |
| $400n$ | $256m$ |
| $2n^2$ | $16m$ |
| $n^5$ | $3.03m$ |
| $2^n$ | $m + 8$ |

# Calcul du nombre d'opérations primitives



**Algorithm:** arrayMax
**Input:** $A$ un tableau de $n$ entiers ($n > 0$)
**Output:** La valeur maximale dans $A$

```
currentMax ← A[0]                          // 2 opérations
for i ← 1 to n − 1 do                       // 1 + 2.(n − 1) + 2.n op
    if currentMax < A[i] then               // 3 op., exécutées n − 1 fois
        currentMax ← A[i]                   // 2 op., exécutées au plus n − 1 fois
return currentMax                           // 1 opération
```

$i{+}{+} \equiv i \leftarrow i+1$ : 2 opérations primitives ; $i \leq n-1$ : 2 opérations primitives

Dans le pire cas (p.ex. $A$ en ordre croissant), $9n-3$ opérations primitives
Dans le meilleur cas (p.ex. $A$ en ordre décroissant), $7n-1$ opérations primitives
On peut négliger les constantes $-3$ et $-1$

Ce calcul introduit de nouvelles constantes (p.ex. $9$ ou $7$) que l'on peut négliger pour les mêmes raisons que précédemment

# Explications



Ce calcul introduit de nouvelles constantes (p.ex. 9 ou 7) que l'on peut négliger pour les mêmes raisons que précédemment

Que se passe-t-il si la taille du problème $n$ est multipliée par 10 ?

$$f(n) = 9n \Rightarrow f(10n) = 9.(10n) = 10.(9n) = 10.f(n)$$

Le temps calcul est multiplié par 10
$\Rightarrow$ la constante multiplicative 9 n'influence pas le rapport $\frac{f(10n)}{f(n)}$

Si l'on s'intéresse à l'influence de la taille du problème sur le temps calcul, on peut négliger cette constante multiplicative