UCLouvain

Algorithmique

Cours préparatoires au master en sciences informatiques

Vincent Branders

2022–2023

Complexité d'algorithmes simples

Qu'est-ce qu'un algorithme ?

Définition informelle

- L'idée derrière un programme
- Tout ce qui ne change pas que le programme soit
  - écrit en Python, Java, C,...
  - exécuté sur un super-calculateur ou sur un smartphone
  - ...ou dans la tête quand on joue par exemple aux cartes

En quoi un algorithme est-il utile ?

Résoudre un problème général et précisément spécifié

- L'ensemble des instances du problème sur lequel l'algorithme doit fonctionner
- Les propriétés que doivent satisfaire les résultats

Un exemple : le problème du tri

Input: Une séquence de n nombres $a_1, \ldots, a_n$ (avec $n \geq 1$)
Output: Une permutation (un réordonnancement) de la séquence d'entrée telle que $a_1 \leq a_2 \leq \cdots \leq a_n$

Exemple :

$9, 45, 22, 5, 37 \implies 5, 9, 22, 37, 45$

$10, 4.5, 3, 0.0, 67, -27.4, 3, 10, -5 \implies -27.4, -5, 0.0, 3, 3, 4.5, 10, 10, 67$

- Il s'agit de 2 instances qui se différencient par les valeurs d'entrée et la taille du problème
- Nous cherchons des algorithmes corrects et efficaces

Exactitude

Pour tout algorithme, nous devons prouver qu'il renvoie toujours un résultat correct pour n'importe quelle instance du problème qui satisfait la spécification

Input: Une séquence de n nombres $a_1, \ldots, a_n$ (avec $n \geq 1$)
Output: Une permutation (un réordonnancement) de la séquence d'entrée telle que $a_1 \leq a_2 \leq \cdots \leq a_n$

Pour le tri, le résultat doit donc être correct même si

- la séquence d'entrée est déjà triée
- la séquence d'entrée contient une ou plusieurs répétitions
- la séquence d'entrée ne contient que des nombres négatifs

...

Portée de la spécification

Input: Une séquence de n nombres $a_1, \ldots, a_n$ (avec $n \geq 1$)

Si l'entrée correspond à une séquence de caractères, c'est en dehors de la spécification initiale

- `b d a c` $\Longrightarrow$ **?**

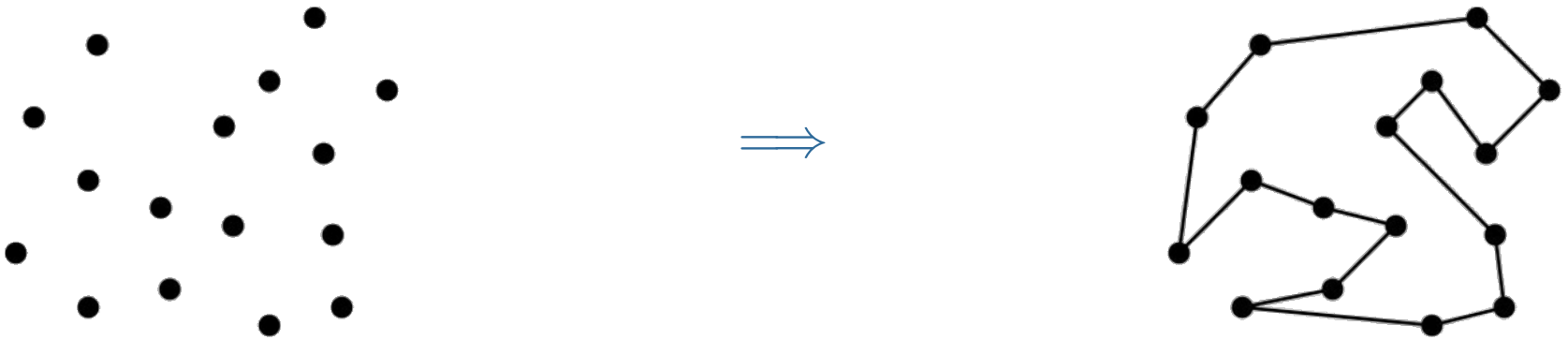
Il serait intéressant que la spécification soit la plus générale possible

Input: Une collection de n éléments que l'on peut ordonner deux à deux ($\leq$)
Output: Un réordonnancement tel que la collection résultante soit en ordre croissant : $a_1 \leq a_2 \leq \cdots \leq a_n$

- `b d a c` $\Longrightarrow$ `a b c d` (selon $\leq$, l'ordre alphabétique)

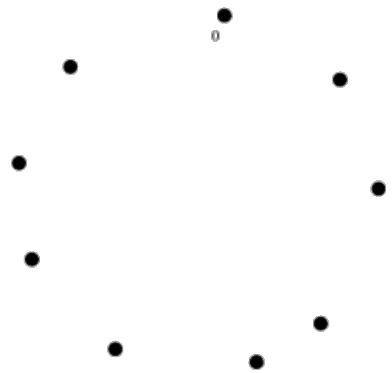
Optimisation d'un bras de robot

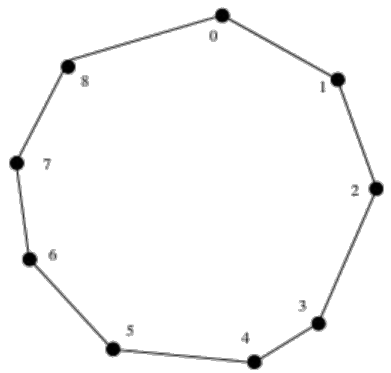
- Vous disposez d'un robot équipé d'un fer à souder
- Vous devez programmer le robot pour qu'il passe par tous les points de contact dans un certain ordre pour souder les contacts
- Nous cherchons l'ordre des points de contact qui minimise le distance parcourue par le bras du robot



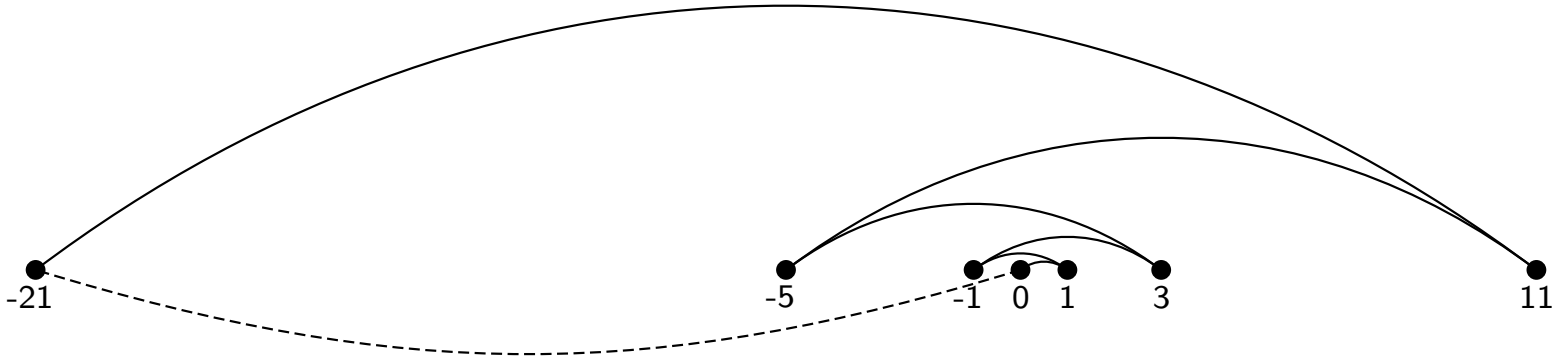
Le tour du plus proche voisin

Algorithm: NearestNeighborTour
Input: Une collection de n points
Output: Un tour passant par tous les points et de distance totale minimale
Visiter un point p_0 initial, choisi au hasard
$i \leftarrow 0$
while *il y a des points non visités* **do**
    $i \leftarrow i + 1$
    Visiter p_i, le plus proche voisin de p_{i-1}, non déjà visité
return *le tour selon lequel les points sont visités*

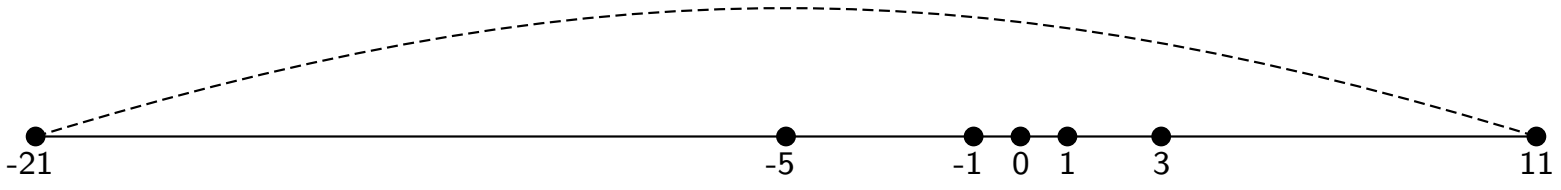




Le tour du plus proche voisin est **incorrect**



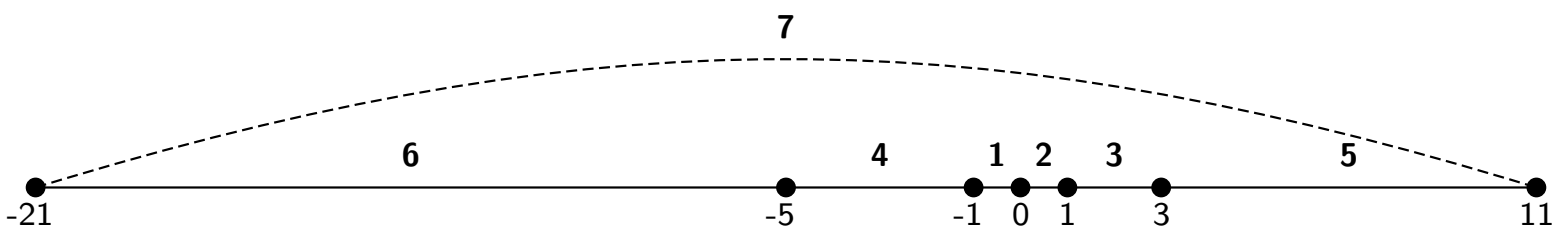
Si nous étions parti du point le plus à gauche, l'algorithme aurait trouvé une solution optimale



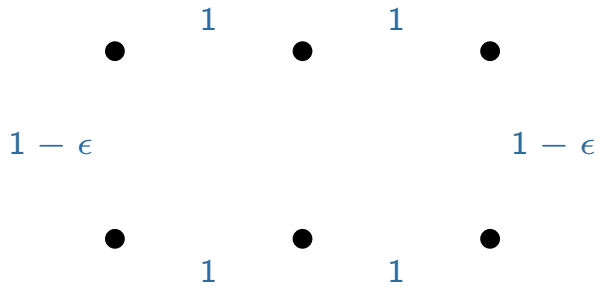
Le tour de la paire la plus proche

Idée : connecter des paires de points proches sans former de branches à 3 arrêtes ou de cycle (sauf pour terminer le tour)

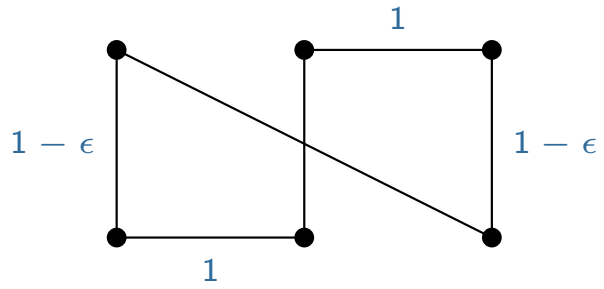
Algorithm: ClosestPairTour
Input: Une collection de n points
Output: Un tour passant par tous les points et de distance totale minimale
for $i \leftarrow 1$ **to** $n-1$ **do**
  $d \leftarrow \infty$
  foreach *paire de points extrêmes* (x, y) *de chemins partiels distincts* **do**
    if $dist(x, y) \leq d$ **then**
      $x_m \leftarrow x; y_m \leftarrow y; d \leftarrow \text{dist}(x, y)$
  Connecter (x_m, y_m) par une arrête
Connecter les deux points extrêmes par une arrête
return *le tour constitué des arrêtes*

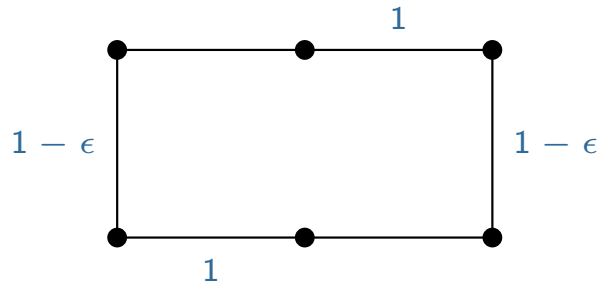


Le tour de la paire la plus proche est **incorrect**



- Les lignes sont légèrement plus proches que les colonnes
- `ClosestPairTour`:
  1. Connecte les lignes verticalement
  2. Connecte les colonnes en alternance
  3. Termine le tour par la diagonale





- La solution de droite est meilleure
  - `ClosestPairTour` ne renvoie pas un tour de distance totale minimale !

Un aglorithme exact : la recherche exhaustive

- Énumérer toutes les permutations possibles de n points
  - Une permutation est un ordonnancements possibles des n points
- Pour chaque permutation, calculer la distance associée
- Enregistrer le tour associé à la permutation de plus petite distance

- Trouver une solution optimale est **garanti**
- L'algorithme est **exact**

La recherche exhaustive est **impraticable**

Le nombre de permutations d'un ensemble à n éléments :

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \cdots \times n$$

- $10! = 1 \times 2 \times 3 \times \cdots \times 10 = 3,628,800$
- $20! \approx 2 \times 10^{18} = 2,000,000,000,000,000,000$
- $100! \approx 9 \times 10^{157}$
- $500! \approx 1 \times 10^{1134}$

Pourquoi pas un super-calculateur ?

Le supercalculateur le plus rapide du monde (en novembre 2022)

- 1.194 exaflops = 1.194.000 millions de millions (1.1×10^{18}) d'opérations par seconde
- Evaluer la distance totale d'un tour requiert au moins 100 opérations si il y a 100 points
- `Frontier` peut évaluer 11.000 millions de millions (1.1×10^{16}) de permutations par seconde

Pourquoi pas un super-calculateur ? (suite)

- C'est sans doute "un peu cher" pour un robot à souder (puissance électrique 21 MW vs. 4kW : 1/5000)
- `Frontier` aurait besoin de 2.6×10^{133} années pour calculer $100! \approx 9 \times 10^{157}$ possibilités

Âge de l'Univers

13 800 000 000
années

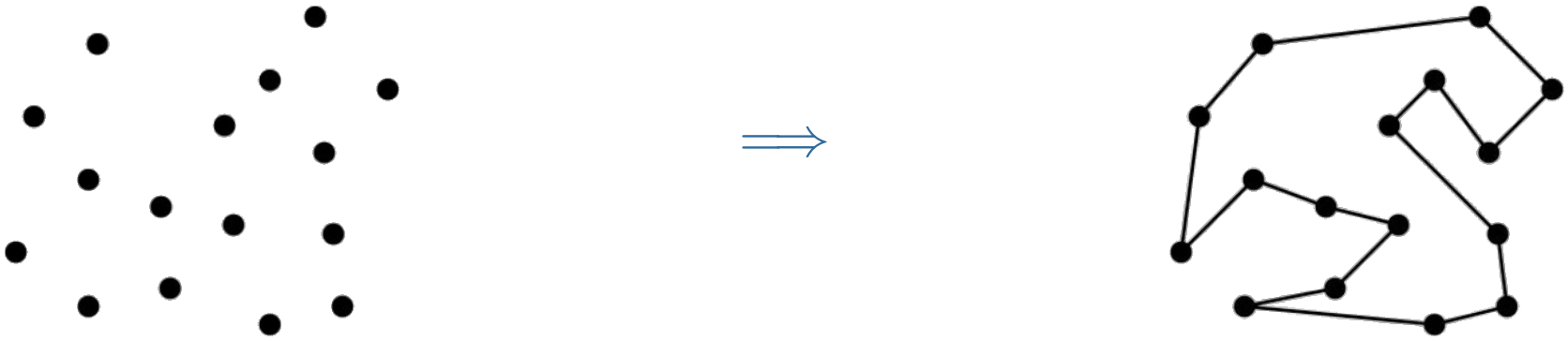
Calcul de Frontier

26 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000
années

Un *algorithme plus rapide* s'exécutant sur un *ordinateur plus lent* finira **toujours** par gagner lorsque la **taille** du problème **augmente**

Le voyageur de commerce

- Le problème du robot à souder est connu sous le nom du voyageur de commerce



  - Passer une fois par chaque ville et minimiser la distance totale parcourue
- Il **n**'existe **pas** d'algorithme exact et efficace pour résoudre ce problème
  - On ne peut le résoudre exactement que pour des instances de très petite taille
  - Pour les instances plus grandes, on utilise des algorithmes n'ayant pas la garantie de trouver une solution optimale

Exactitude ne correspond pas toujours à optimalité

- Un problème n'est pas toujours formulé comme un problème d'optimisation
  - Exemple : un algorithme de tri est exact s'il renvoie toujours une séquence triée

Mais

- on peut souvent changer la formulation :

$$1 \leq 3 \leq 3 \not\leq 2 \leq 7$$

- tri $\equiv$ minimiser le nombre de $\not\leq$:

$$1 \leq 2 \leq 3 \leq 4 \leq 7$$

Prouver qu'un algorithme est **inexact**

Recherche d'un contre-exemple

- Un algorithme doit toujours marcher (par rapport à sa spécification)
- Il suffit d'un contre-exemple pour prouver qu'il est inexact
  - penser à des exemples de petites tailles d'abord
  - penser à des cas d'égalité dans la décision ou l'ordre des opérations (ex. choisir "le" point le plus proche)
  - penser à des cas limites (ex. très grands, très petits, collection vide, collection déjà triée, collection triée dans l'ordre inverse, uniquement des valeurs négatives,...)

⚠ Ne pas trouver de contre-exemple
ne constitue **pas** une preuve que l'algorithme est exact !

À propos de la Pierre de Résurrection

Prouvez-moi plutôt qu'elle n'existe pas.

Xenophilius Lovegood

Enfin, voyons... je suis désolée, mais c'est complètement ridicule ! [...]
Si on va par là, on peut affirmer que toute chose existe s'il suffit pour y croire que personne n'ait jamais réussi à démontrer qu'elle n'existait pas !

Hermione Granger

Oui, on peut. Je suis content de voir que votre esprit commence à s'ouvrir.

Xenophilius Lovegood

...non, on ne peut pas !

Prouver qu'un algorithme est exact et efficace

Preuve d'exactitude

Plus difficile, pas de recette universelle...

Preuve d'efficacité

- Éviter un temps calcul rédhibitoire lorsque la taille du problème augmente
  - on s'intéresse au passage à l'échelle
  - pas besoin d'un programme pour trier 10 nombres, mais utile pour 50,000 !
- La notion d'efficacité est relative par rapport aux différents algorithmes qui résolvent le problème
- Nous y reviendrons fréquemment...

Caractériser l'efficacité d'un programme

Caractériser l'efficacité d'un programme ?

Grandeurs

- Le temps calcul que met le programme à produire un résultat
  $\Rightarrow$ lien avec la complexité temporelle de l'algorithme
- L'espace utilisé (mémoire, disque) par le programme
  $\Rightarrow$ lien avec la complexité spatiale de l'algorithme

$\Rightarrow$ Détermine quels problèmes peuvent être résolus

Quels sont les facteurs influençant le temps (ou l'espace) ?

Étude d'un algorithme

Algorithm: paireMax
Input: A, un tableau de $n(\geq 2)$ entiers
Output: La somme maximale obtenue à partir d'une paire de A

```
sommeMaximale ← −∞
for i ← 0 to n − 1 do
    for j ← i + 1 to n − 1 do
        sommeCourante ← A[i] + A[j]
        if sommeCourante > sommeMaximale then
            sommeMaximale ← sommeCourante
return sommeMaximale
```

Note :

x to y comprend les valeurs de l'intervalle $[x, y]$

x until y comprend les valeurs de l'intervalle $[x, y[$

x downto y comprend les valeurs de l'intervalle $[y, x]$

- Qu'est-ce qui influence le temps d'exécution ?
- Qu'est-ce qu'on contrôle ?

Wooclap !

https://app.wooclap.com/ANQSFP

Les facteurs influençant le temps d'exécution

- Les **données** du problème (l'instance particulière : taille + valeurs)
- L'**algorithme** utilisé pour résoudre le problème

mais aussi...

- Le *matériel* (vitesse du processeur, taille et vitesse d'accès à la mémoire, temps de transfert disque,...)
- Le *logiciel* (langage de programmation, compilateur/interpréteur,...)
- La *charge de la machine* (nombre de processus qui s'exécutent,...)
- Le *système d'exploitation* (gestion des différents processus,...)
- La *charge du réseau* (accès aux données, écriture des résultats,...)

Algorithm: paireMax
Input: A, un tableau de $n(\geq 2)$ entiers
Output: La somme maximale obtenue à partir d'une paire de A

```
sommeMaximale ← −∞
for i ← 0 to n − 1 do
    for j ← i + 1 to n − 1 do
        sommeCourante ← A[i] + A[j]
        if sommeCourante > sommeMaximale then
            sommeMaximale ← sommeCourante
return sommeMaximale
```

- La **taille** du problème (ici, la taille n du tableau)
- Les **valeurs spécifiques** définissant une **instance** particulière du problème (ici, les valeurs mémorisées dans le tableau)
- La taille du problème **est** le nombre de valeurs à spécifier pour définir une instance particulière du problème

Meilleur cas, pire cas ou cas moyen

- Il y a souvent un nombre infini d'instances possibles (ici, toutes les valeurs possibles d'un tableau de taille n contenant des entiers)
- 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)
- 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

Mesure expérimentale du temps d'exécution

Procédure

1. Écrire un programme implémentant l'algorithme à étudier
2. Exécuter le programme pour différentes instances du problème (taille + valeurs spécifiques)
3. Utiliser une méthode pour mesurer le temps effectif d'exécution
   - Python : `time.process_time()`
   - Java : `System.currentTimeMillis()`

   ...

```
import time
start = time.process_time()
# ...
# your python code here
# ...
stop = time.process_time()
print("CPU time = {0:.2E} sec".format(stop - start))
```

Paire maximale

Algorithm: paireMax
Input: A, un tableau de $n(\geq 2)$ entiers
Output: La somme maximale obtenue à partir d'une paire de A

```
sommeMaximale ← −∞
for i ← 0 to n − 1 do
    for j ← i + 1 to n − 1 do
        sommeCourante ← A[i] + A[j]
        if sommeCourante > sommeMaximale then
            sommeMaximale ← sommeCourante
return sommeMaximale
```

- Réaliser une mesure expérimentale du temps d'exécution
- Faire varier : → le nombre de valeurs à spécifier pour définir une instance particulière du problème
  - La taille du problème (ici, la taille n du tableau)
  - Les valeurs spécifiques définisssant une instance particulière du problème (ici, les valeurs mémorisées dans le tableau)

À faire

- Comparer les graphes obtenus individuellement
- Étudier l'influence de n
  - Temps constant, linéaire, quadratique,...?
- Les graphes sont-ils identiques entre étudiants ?
- Les implémentations sont-elles très similaires ou plutôt différentes ?

(Les résultats sont-ils très différents si on utilises son ordinateur en parallèle ?)

Penser à automatiser la production de graphes : on va le faire souvent...

Un autre algorithme pour résoudre le même problème

Algorithm: paireMaxAlt
Input: A, un tableau de $n(\geq 2)$ entiers
Output: La somme maximale obtenue à partir d'une paire d'entiers de A

$max \leftarrow [0, 1]$
if $A[1] > A[0]$ **then**
  $max \leftarrow [1, 0]$
for $i \leftarrow 2$ **to** $n - 1$ **do**
  if $A[i] > A[max[1]]$ **then**
    $max[1] \leftarrow i$
    if $A[max[1]] > A[max[0]]$ **then**
      $max = [max[1], max[0]]$
return $A[max[0]] + A[max[1]]$

- Réaliser une étude expérimentale du temps d'exécution
- Comparer le graphe du temps en fonction de n avec le graphe produit pour `paireMax`

Limites de l'approche expérimentale ?

- Nécesite d'implémenter les différents algorithmes que l'on veut comparer

  (sans compter la diversité des implémentations pour un algorithme unique)
- Nombre limité (et forcément fini) d'instances testées
- Ces instances ne sont pas forcément représentatives de tous les cas
- Outre l'algorithme et les instances testées, tous les autres facteurs (logiciel, matériel, ...) influencent la mesure du temps d'exécution

$\Rightarrow$ Besoin d'une meilleure façon de caractériser l'efficacité d'un algorithme

Un autre problème

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
```

- Comparer `firstNegative` et `firstNegativeRev`

Un autre problème 2

Algorithm: deuxMaxSomme
Input: A, un tableau de $n(\geq 4)$ entiers
Output: La somme maximale obtenue à partir d'une paire d'entiers de A

```
max ← [0, 1]
if A[1] > A[0] then
    max ← [1, 0]
for i ← 2 to n − 1 do
    if A[i] > A[max[1]] then
        max[1] ← i
        if A[max[1]] > A[max[0]] then
            max = [max[1], max[0]]
return A[max[0]] + A[max[1]]
```

Algorithm: quatreMaxSomme
Input: A, un tableau de $n(\geq 4)$ entiers
Output: La somme maximale obtenue à partir de quatre entiers de A

```
max ← [0, 1, 2, 3]
// Bubble-sort max
for i ← 1 to 3 do
    j ← i
    while j ≥ 1 and A[max[j]] > A[max[j − 1]] do
        tmp ← max[j]
        max[j] ← max[j − 1]
        max[j − 1] ← tmp
        j ← j − 1
for i ← 5 to n − 1 do
    j ← 3
    if A[i] > A[max[j]] then
        max[j] ← i
        while j ≥ 1 and A[max[j]] > A[max[j − 1]] do
            tmp ← max[j]
            max[j] ← max[j − 1]
            max[j − 1] ← tmp
            j ← j − 1
return A[max[0]] + A[max[1]] + A[max[2]] + A[max[3]]
```

- Comparer `deuxMaxSomme` et `quatreMaxSomme`

⚠Les problèmes sont différents

Un autre problème 3

Algorithm: max
Input: A, un tableau de n entiers
Output: Renvoie le maximum de A

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

Algorithm: max2
Input: A, un tableau de n entiers
Output: Renvoie le maximum de A

```
sorted ← sort(A)
return sorted[n − 1]
```

- Comparer `max` et `max2`

Résumé

- Un algorithme est l'idée derrière un programme
- Un algorithme est indépendant du langage de programmation, du système d'exploitation
- Un algorithme est indépendant de la machine sur laquelle s'exécute le programme qui implémente l'algorithme
- Un bon algorithme
  - a une portée suffisamment large
  - est exact
  - est efficace
- Pour certains problèmes (même très courants et simples à énoncer), il n'existe pas d'algorithme exact et efficace
- Pour d'autres problèmes, il existe plusieurs algorithmes exacts
  - certains sont plus efficaces que d'autres

...et donc ?

On souhaite caractériser l'efficacité d'un algorithme *exact*

- indépendemment de l'implémentation ou de la machine qui l'exécute
- en tenant compte des particularités observées dans les problèmes précédants
  - `firstNegative` et `firstNegativeRev`
    - influence du choix d'instances
  - `deuxMaxSomme` et `quatreMaxSomme`
    - influence de la taille des instances
  - `max` et `max2`
    - influence des différentes opérations effectuées

$\Rightarrow$ besoin de caractériser les différents types d'instances possibles