Algorithmique et Analyse d'Algorithmes

L3 Info

Cours 1 : notion de coût d'un algorithme

Benjamin Wack

2021 – 2022

Plan

Objectifs du cours

Savoir proposer une solution algorithmique à un problème posé, savoir implanter la solution et savoir analyser celle-ci.

Objectifs détaillés

- Savoir **reconnaître** et mettre en œuvre des **schémas génériques** d'algorithmes (séquence, arbre, graphe...),
- Savoir **construire** une solution selon une démarche allant du plus simple (algorithme naïf) au plus efficace (diviser pour régner, etc.)
- Savoir **démontrer** la correction des algorithmes
- Savoir comment **évaluer** la complexité d'une solution algorithmique :
  - analyser la complexité au pire, en moyenne avec des **hypothèses probabilistes**,
  - analyser la complexité en utilisant des mesures sur des simulations ou des **jeux de test**.

Structure du cours

Cours (B. Wack)

Une partie synthétique sur les concepts et schémas algorithmiques
Un algorithme classique afin de se constituer une culture de référence

TD1 (V. Garnero, A. Rasse, J.-M. Vincent, B. Wack)

Exercices permettent de renforcer la compréhension des concepts.

TD2 (V. Danjean, L. Gourdin, A. Jacquier, D. Monniaux)

Mise en œuvre des concepts et préparation aux activités pratiques

APNEES : Activités Personnelles Non Encadrées Évaluées

Validation des concepts par la pratique et évaluation de la compréhension

+ le travail personnel ! (exercices, petits programmes...)

Adresse Mail enseignant : `Prénom.Nom@univ-grenoble-alpes.fr`

Ressources

Bibliographie

- *Algorithmique*, T. Cormen, R. Rivest & C. Leiserson, Dunod
- *Algorithmes*, R. Sedgewick, Pearson Education

Page Web

Planning, documents, annales

`https://algo.gricad-pages.univ-grenoble-alpes.fr/L3I-S5-algo`

(ou par ma page Web)

Moodle de l'UFR

Sujets et rendus d'Apnées

`https://im2ag-moodle.univ-grenoble-alpes.fr/course/view.php?id=229`

Évaluations

Note finale = 65 % Examen + 35 % CC

Les contrôles continus

- 2 quicks (début octobre et début novembre) : 10 % chacun
- 3 comptes-rendus d'APNEEs : 15 % au total

Examen terminal

- 2h30 sans documents ni calculatrice
- Session 2 en juin...

Challenge de programmation

- Semaine du 8 au 12 novembre
- En équipes

Programme (indicatif) du cours

- **Complexité des algorithmes**
  1. Coût d'un algorithme (itérations, ordres de grandeur) — *La star*
  2. Analyse en moyenne — *Quicksort*
- **Preuves d'algorithmes**
  3. Invariant, correction, terminaison — *Drapeau hollandais*
  4. Logique de Hoare — *Dichotomie*
- **Types abstraits élémentaires et implantation**
  5. Structures séquentielles — *Algorithme de parenthésage*
  6. Structures arborescentes — *Partition Binaire de l'Espace*
- **Arbres**
  7. Arbres binaires de recherche — *Arbres B*
  8. Arbres ordonnés, structure de tas
  9. Arbres et codage — *Algorithme de Huffman*
- **Graphes et algorithmes**
  10. Algorithmes gloutons — *Coloration de graphes*
  11. Algorithmique de graphes — *Prim et Kruskal*

Notion de **problème** ***données***

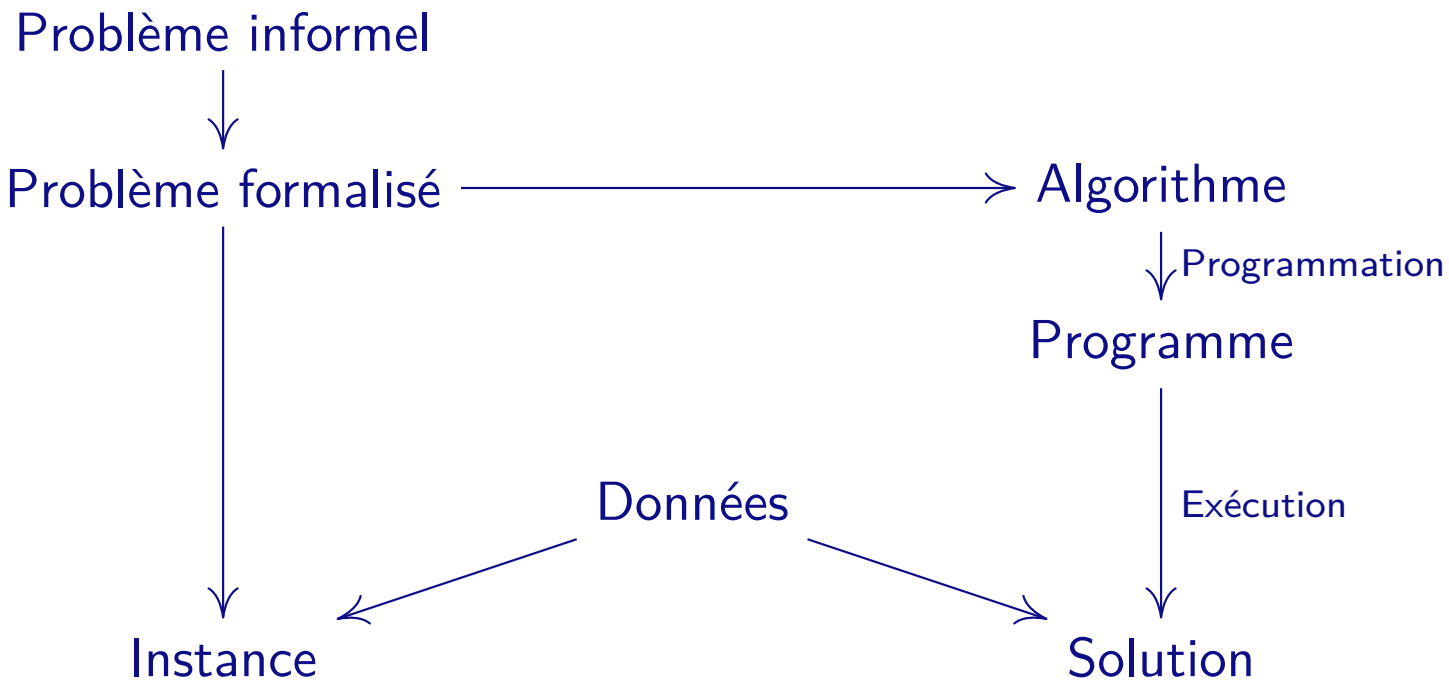
- Effectuer une recherche dans un inventaire *stock*
- Résoudre une équation *coefficients*
- Trouver le plus court chemin sur un plan de ville *plan*
- Corriger des fautes d'orthographe *texte + dictionnaire*
- Multiplier des matrices *coefficients*
- ...

Dans chacun de ces exemples on s'intéresse en fait à une classe de problèmes similaires, dont chaque instance est définie par des données.

Il faut également préciser le résultat attendu.
(par exemple pour l'inventaire : oui/non ? quantité ? localisation ?)

Qu'est-ce qu'un algorithme ?

Procédure de résolution de **n'importe quelle instance** d'un problème suffisamment élémentaire pour être exécutée de façon automatique.



Exemple : plus court chemin

Un algorithme traite donc des données pour produire un résultat ; mais avec quelles primitives et quelles ressources ?

S'il fallait tenir compte :

- du matériel
- du système d'exploitation
- du langage de programmation

on passerait son temps à réinventer les mêmes algorithmes.

Algorithme versus programme

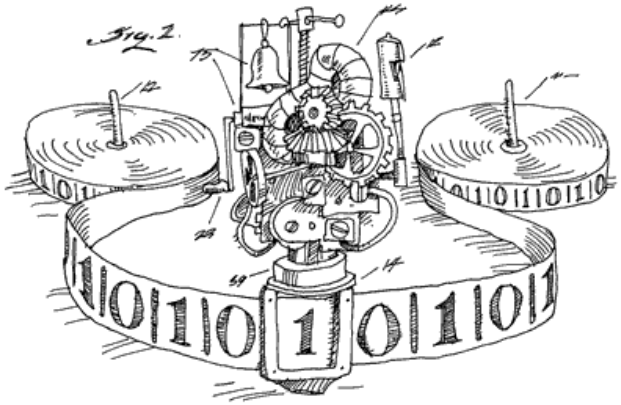
Un algorithme décrit une méthode générale de résolution d'un problème :

- que l'on pourra traduire dans n'importe quel langage de programmation
- que l'on pourra analyser pour en dégager les caractéristiques

Contre-exemples : tri spaghetti

Nécessité d'un **modèle**

Alan M. Turing
1912-1954



Machine « de papier »
1936

La machine est dotée :

- d'un ruban infini ($\simeq$ mémoire)
- d'une tête de lecture/écriture
- d'un automate ($\simeq$ processeur)

Un modèle donc tourné vers les opérations de lecture/écriture ($\simeq$ affectation, calculs)

Les questions "résolues" par Turing

- Qu'est-ce qu'un calcul effectué automatiquement ?
- Peut-on tout calculer ?
- Peut-on décider automatiquement si une formule logique est vraie ?

10 ans avant les premiers ordinateurs !

Cf. *Modèles de calcul* au semestre 6

Notre modèle de machine

Une machine est constituée d'un processeur, d'une **mémoire** et d'un ensemble d'**opérations**.

- **Mémoire**
  - infinie (mais un calcul donné utilise un espace fini),
  - types élémentaires (finis) `int, float,...`
  - puis on ajoutera des structures de données
- **Opérations**
  - nombre fini d'opérations (arithmétiques, booléennes...) ;
  - chaque opération a un nombre fini de paramètres (nombre fini de variables lues et écrites)
- **Processeur**
  - processeur unique ;
  - effectue les opérations en temps constant (1 top = 1 opération)

Les questions traitées en Algo5

Est-ce que je peux construire un programme qui résout mon problème (sur une machine conforme à mon modèle) ?

Spécification du problème et construction d'un algorithme

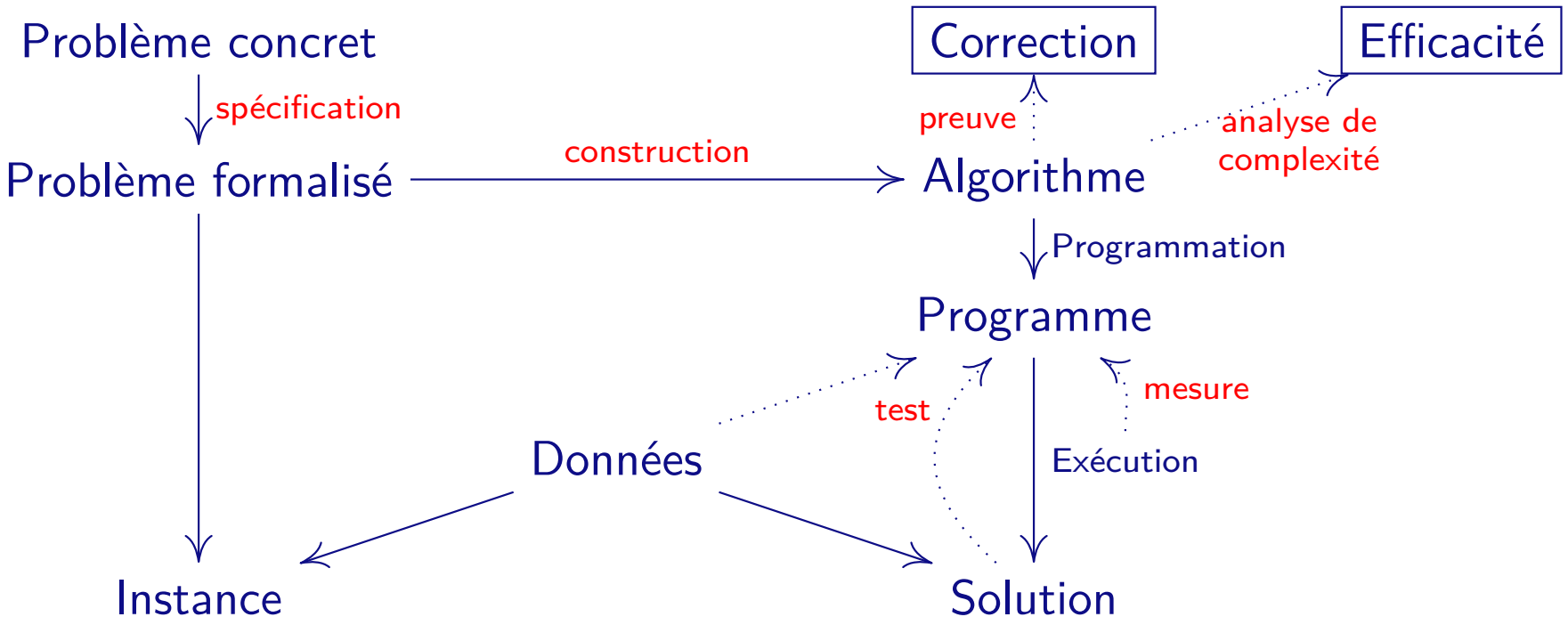
Est-ce que l'exécution de mon programme sur la machine donne bien le résultat souhaité ?

Vérification de propriétés qualitatives par test et preuve

Est-ce que mon programme me fournit le résultat en un temps acceptable ?

Validation de propriétés quantitatives par mesure et analyse

Les questions traitées en Algo5



Exemple : plus court chemin

Efficacité d'un algorithme

- Est-ce que mon programme consomme une quantité **acceptable** de ressources pour fournir un résultat ?
- Étant donnés deux programmes résolvant le même problème, lequel est le "**meilleur**" ?

Notion(s) de coût(s)

- coût en temps
  = nombre d'opérations permettant d'obtenir le résultat à partir des données
- coût en mémoire
  = nombre maximal de variables utilisées simultanément durant le calcul

Exemple (1)

Maximum de 3 éléments

MAX(a,b,c)
Données : Trois éléments de même type (comparable) : a, b et c
Résultat : Le plus grand des trois

if $a > b$
  $m := a$
else
  $m := b$
if $c > m$
  $m := c$
`Return` (m)

Coût de l'algorithme

Coût temps ? 2 comparaisons + 1 ou 2 affectations
Coût en espace mémoire ? 1 variable

Coût constant, indépendant des données

Exemple (2)

Comptage d'un élément

OCCURRENCES(x, T)
Données : Un élément x et un tableau T de taille n
Résultat : Le nombre d'occurrences de x dans T

$k := 0$
for $i := 1$ **to** n
  if $T[i] = x$
    $k := k + 1$
`Return` (k)

- Coût en temps : n comparaisons
  (+ les opérations sur les indices)
- Coût variable, dépend de la **taille** des données
  mais toujours n comparaisons pour un tableau de taille n

Exemple (3)

Maximum d'un tableau

MAXIMUM(T)
Données : T tableau de n éléments
Résultat : L'indice d'un élément maxi

```
max := T[1]
for i := 1 to n
    if T[i] ≥ max
        i_max := i
        max := T[i]
        Traiter (max)
Return (i_max)
```

Coût de l'algorithme

Modèle de coût ?
Supposons `Traiter` coûteux

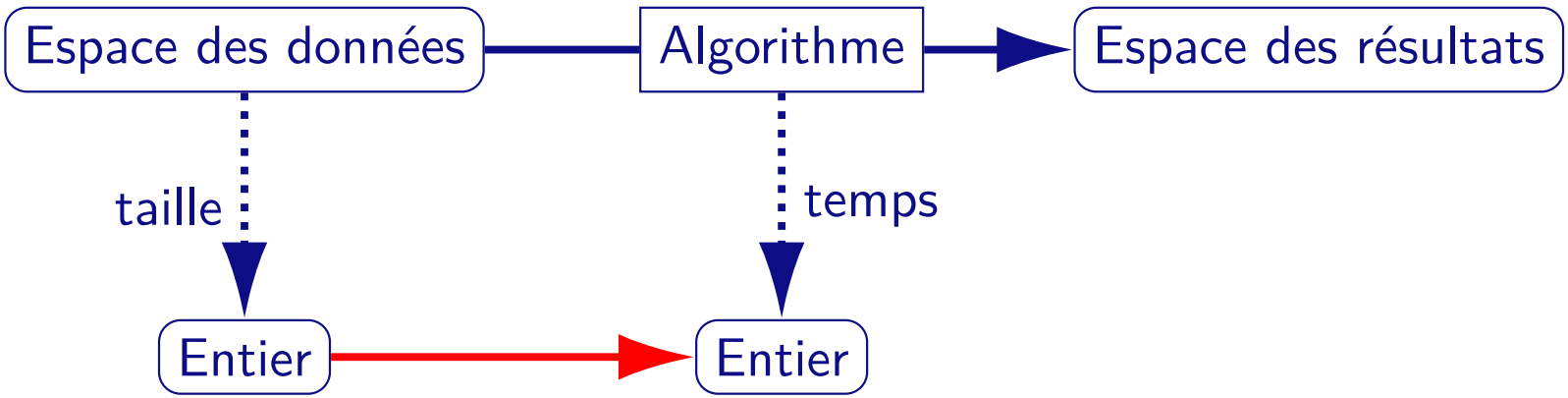
- pour T=[1,2,...,10] ? 10
- pour T=[1,2,...,n] ? n
- pour T=[n,1,2,...,n-1] ? 1

Dépend de la **taille** des données
Dépend aussi de la **valeur** des données

Le coût calculé doit avoir une valeur *générale*

- Il s'exprime toujours en fonction de la *taille* des données
- Si trop de variation on cherche une *borne supérieure* (pire cas)

Généralisation



Caractériser l'efficacité générale de l'algorithme =
trouver la relation entre la **taille des données** et le **coût de l'algorithme** sur un (modèle de) machine donnée

Remarque : attention aux entiers, le temps de calcul de x^n dépend de n qui n'est pas la taille de la donnée.

Coût d'un algorithme

Coût (au pire)

Le coût d'un algorithme $\mathcal{A}$ est **fonction** de la **taille** des données :

$$C_{\mathcal{A}}(n) = \max(\text{coût}(d)) \text{ pour toutes les données } d \text{ de taille } n$$

- Suppose d'avoir fixé la notion de taille
- Maximum = garantie quelles que soient les conditions d'utilisation
- Concrètement : majorer le coût + exhiber un cas défavorable

Coût au mieux

$$C_{\mathcal{A}}^{min}(n) = \min(\text{coût}(d)) \text{ pour toutes les données } d \text{ de taille } n$$

- Correspond au cas le plus favorable

Quel comportement privilégie-t-on ?

Coûts des structures de base

Instructions en séquence

Le coût de
Faire Truc
Faire Machin
est la somme des coûts de Truc et de Machin

Composition des coûts

Le coût d'une boucle est donc la somme
des coûts de chaque itération

Instructions **FOR, WHILE, REPEAT,...**

Coûts des structures de base (2)

Instructions conditionnelles

if *condition*
  Faire Truc
else
  Faire Machin

Coût de *l'une* des branches plus le coût d'évaluation de la condition

Majoration du coût

Le coût au pire sera donc majoré par le maximum des coûts au pire de chaque branche.

$$Cout(\textit{si Condition alors A sinon B}) \leq Cout(evaluation(Condition)) + \max\{Cout(A), Cout(B)\}$$

Instructions **IF, SWITCH,...**

Coûts des structures de base (3)

Appel de procédure

Le coût de l'appel d'une procédure est :

- le coût du corps de la procédure pour ses paramètres d'appel
- plus le coût de l'évaluation de ses paramètres.

Méthode de calcul de coût

Le calcul du coût d'un algorithme s'obtient donc en **composant** les coûts des différentes opérations composant l'algorithme.

- assemblage et reconstruction
- méthode par composition
- se fait à la **conception de l'algorithme**

Exemple

Calcul de la somme des entiers de 1 à n

```
i := 1
somme := 0
Tantque (i ≤ n) :
    somme := somme + i
    i := i + 1
```

- Coût d'une itération = 2 (additions) + 1 test
- Nombre d'itérations = n
- Coût de la boucle = $3n$
- Coût de l'algorithme = $3n + 1$

Niveau de détail judicieux ?

Ordres de grandeur

- La complexité est une prédiction du temps d'exécution du programme codant l'algorithme.
- Mais ce temps dépend de l'architecture de la machine, donc c'est une abstraction (approximation)
- On s'intéresse au passage à l'échelle des algorithmes plus qu'à une mesure précise du temps d'exécution

Ce qui est important c'est :

1. l'ordre de grandeur ;
2. de pouvoir comparer les algorithmes

Complexité d'un algorithme

On appelle **complexité** d'un algorithme une fonction de référence (logarithme, polynome, exponentielle...) comparable à son coût.

Exemples — $n = 10^6$

- moyenne des éléments d'un tableau de taille n
  $\Rightarrow$ complexité en n opérations
  $1/1000^{\text{e}}$ s
- tri par insertion des éléments d'un tableau de taille n
  $\Rightarrow$ complexité en n^2 opérations
  $1/4$ h
- énumération des vecteurs de bits de taille n
  $\Rightarrow$ complexité en 2^n opérations
  $10^{300\ 000}$ années

Objectif : définir des ordres de grandeur comparables

Pour se donner une représentation concrète : sur un PC récent, environ 1 milliard d'opérations / seconde

Borne supérieure asymptotique

Notation $\mathcal{O}$

Pour une fonction donnée g on note $\mathcal{O}(g)$ l'ensemble de fonctions :

$$\mathcal{O}(g) = \{f \text{ telles que } \exists c \geq 0, \exists n_0 \geq 0, \forall n \geq n_0, \quad f(n) \leq c.g(n)\}$$

On écrit $f = \mathcal{O}(g)$ pour $f \in \mathcal{O}(g)$.
On dit que g est une borne supérieure asymptotique pour f.

Vite dit : *f est dépassée par g à partir d'une certaine taille de données*

Pause courbes

Exemples

$$n = \mathcal{O}(n^3) \qquad 1250n^3 = \mathcal{O}(2^n)$$

$$n + \sqrt{n} = \mathcal{O}(n) \qquad n\sqrt{n} + n\log n = \mathcal{O}(n\sqrt{n})$$

Échelles de comparaison (à connaître)

Échelle polynomiale

- Si $p \leq q$ alors $n^p = \mathcal{O}(n^q)$

Échelle logarithmique

- $\log(n) = \mathcal{O}(n)$

Échelle exponentielle

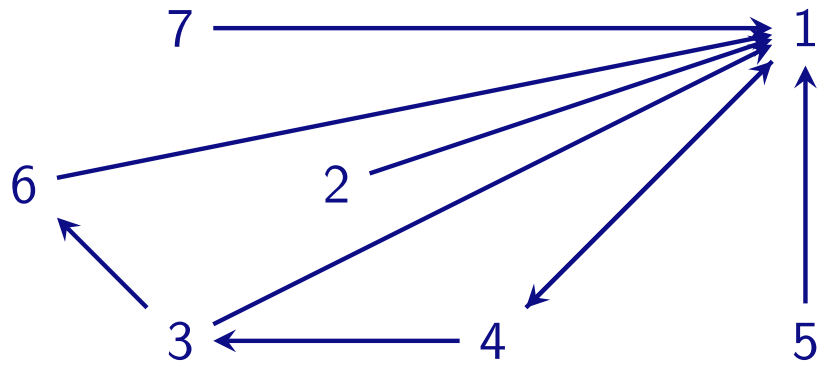
- Si $0 < a \leq b$ alors $a^n = \mathcal{O}(b^n)$
- Pour tous $a > 1$ et $p \geq 0$ on a $n^p = \mathcal{O}(a^n)$

Deux propriétés utiles

Si $f = \mathcal{O}(g)$ alors $f + g = \mathcal{O}(g)$ et $k \times f = \mathcal{O}(g)$

Problème : trouver la star

Soit un groupe de n personnes numérotées de 1 à n.
Une personne i *connaît* j ou bien elle ne la connaît pas.



Une star est une personne :

- que tout le monde connaît
- mais qui ne connaît personne

Précisons les choses :
Résultat : Y a-t-il une star, et si oui quel est son numéro ?

Données : L'entier n, qui connaît qui

Modèle de coût :
On comptera le nombre de questions « *est-ce que i connaît j ?* »

Algorithme naïf

naïf = lent mais assez simple pour se persuader qu'il est correct

```
STAR_NAIF(n)
for i := 1 to n
    i_star := true
    for j := 1 to n                                  }
        if j ≠ i et j ne connaît pas i               } (i est-elle connue de tous ?)
            i_star := false
    for j := 1 to n                                  }
        if j ≠ i et i connaît j                      } (i connaît-elle quelqu'un ?)
            i_star := false
    if i_star
        Return (i)
Return (« pas de star »)
```

Complexité en $\mathcal{O}(n^2)$

Faire mieux

Peut-on accélérer cet algorithme ?

Factoriser les 2 boucles **for** j en une seule

Non : on économise quelques comparaisons (entre indices) mais on ne pose pas moins de questions.

Remplacer les boucles **for** par des boucles **while**

Non plus : on gagne quelques questions en général, mais dans le pire des cas on reste en $\mathcal{O}(n^2)$.

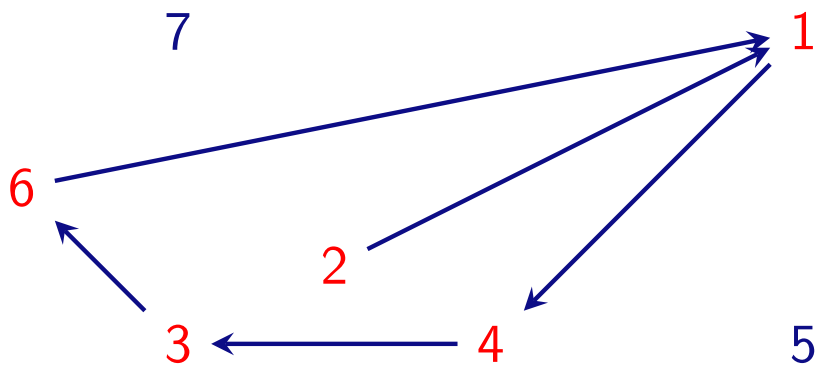
Mieux exploiter les réponses aux questions

Oui :

- Éviter de poser deux fois la même question
- Si i connaît j, alors i n'est pas une star

Une tentative

Idée : suivre les liens « ... connaît ... ».



Mais après... ? ? ?

La priorité : résoudre le problème

Mieux vaut un algorithme :

- lent mais correct
- que « optimisé » mais qui fait n'importe quoi...

Reprenons calmement

- Si i connaît j, alors i n'est pas une star
- Si i ne connaît pas j, alors j n'est pas une star

On devrait donc pouvoir éliminer un candidat à chaque question.

```
STAR_EFFICACE(n)
i := 1 // Candidat star

j := 2 // Prochain candidat à éliminer

while j ≤ n
    if j connaît i          // j n'est pas une star, i peut-être
        j := j + 1
    else                    // i n'est pas une star, j peut-être
        i := j
        j := j + 1
// Puis vérifier (comme avant) si i est bien une star
```

Complexité en $\mathcal{O}(n)$

Conclusion

- Un même problème peut être résolu par des algorithmes très différents
- On peut (parfois) passer de l'un à l'autre par raffinement
- L'analyse de complexité est un critère fiable pour les comparer
- ... mais pas le seul

La prochaine fois

- schémas récursifs
- analyse du coût en moyenne
- tri rapide