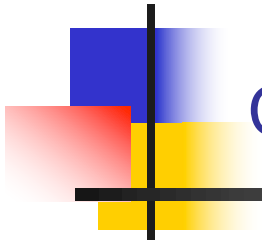
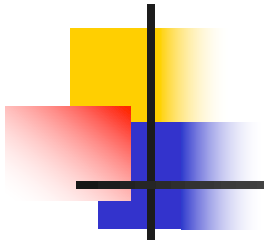


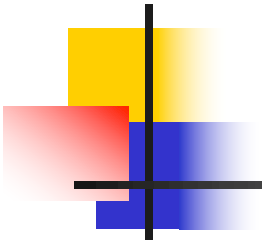
Chapitre 1

Introduction à l'algorithmique



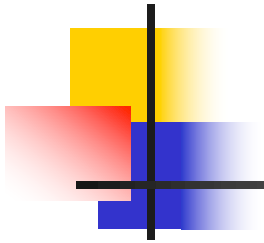
Le concept d'algorithme

- Un algorithme est une séquence d'instructions non ambiguës permettant de résoudre un problème
  - C'est donc une procédure pour obtenir une solution et non pas la solution elle-même
  - Il doit donc exister quelqu'un ou quelque chose (ex: un ordinateur) capable d'effectuer la séquence d'instructions
  - Chaque étape, ou instruction, doit être non ambiguë: elle ne doit pas être un sous algorithme non spécifié
  - Nous devons obtenir la solution en un temps fini pour toute instance possible du problème.
    - L'ensemble des instances possibles doit donc être clairement spécifié
  - Nous désirons des algorithmes efficaces : utilisant peu d'espace mémoire et donnant rapidement une solution



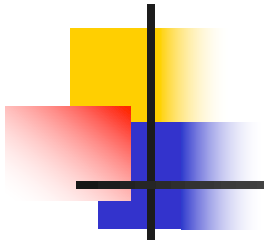
L'importance de l'algorithmique

- L'algorithmique, c'est-à-dire l'étude des algorithmes, est donc pertinente à tous les domaines où l'on rencontre des problèmes pour lesquels on désire obtenir efficacement une solution.
  - Une pertinence couvrant donc l'informatique, les mathématiques, les sciences, le génie, les sciences économiques et de gestion…
- L'algorithmique: c'est le cœur de l'informatique!
  - Qu'est-ce qu'un informaticien sans notions d'algorithmique?
- C'est un domaine fascinant et d'une richesse inouïe:
  - Certains problèmes (comme le tri) admettent plusieurs algorithmes efficaces, mais utilisant des approches différentes
  - Certains problèmes sont NP-complets : ils admettent (probablement) uniquement des algorithmes inefficaces (chap 10)
  - La majorité des problèmes numériques ne peuvent pas être résolus exactement (ex: trouver la valeur exacte de π)
  - Certains problèmes sont indécidables (voir chap 10): il n'existe aucun algorithme pouvant fournir une solution (qui existe)



Comment passer du problème à l'algorithme

- Examinons ce qu'il faut faire généralement pour passer d'un problème à un algorithme permettant de le solutionner
- Les grandes étapes (dans l'ordre):
  - Définir précisément le problème en cause
  - Identifier l'entité (la machine) qui exécutera l'algorithme
  - Choisir entre une solution exacte ou approximative
  - Choisir les structures de données appropriées
  - Concevoir l'algorithme
  - Décrire l'algorithme
  - Prouver que l'algorithme est correct
  - Analyser l'algorithme pour déterminer son efficacité
  - Coder l'algorithme



Définir précisément le problème en cause

- Il faut précisément définir l'objectif du problème: ce que l'on cherche à obtenir ou à optimiser
  - Exemple que nous utiliserons ici: trouver le plus grand commun diviseur entre deux entiers non négatifs m et n

- Il faut spécifier précisément l'ensemble de toutes les instances possibles du problème que l'algorithme devra traiter
  - Ici c'est l'ensemble de toutes les paires (m,n) d'entiers non négatifs
    - Sans perte de généralité (s.p.d.g.), nous allons nous limiter au cas où $m \geq n$.
    - Le plus grand des entiers est positif: $m > 0$.

- Si cette étape est escamotée, notre algorithme pourrait échouer sur une ou plusieurs instances « frontière »

Identifier l'entité (ou machine) qui exécutera l'algorithme

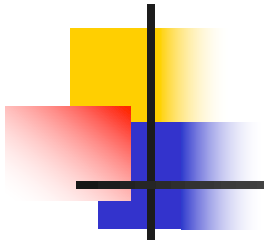
- Nous utiliserons, bien sûr, un ordinateur conventionnel pour exécuter le programme final.
- Nous verrons au chapitre 2 que ce qui importe pour l'algorithme c'est le modèle de calcul théorique sous-jacent à l'ordinateur conventionnel: la machine séquentielle à accès aléatoire (Random Access Machine: «RAM» : 1 processeur, une instruction à la fois).
  - Limitation: cette machine exécute chaque instruction séquentiellement
  - Durant ce cours, nous étudierons uniquement les algorithmes destinés à cette machine séquentielle RAM

- Cette limitation d'exécution séquentielle ne s'applique pas aux ordinateurs à processeurs multiples
  - Nous ne discuterons pas les algorithmes parallèles : ceux qui exploitent le fait que plusieurs instructions peuvent être exécutées simultanément.

Choisir entre une solution exacte ou approximative

- Heureusement, pour plusieurs problèmes « classiques » (ceux que nous rencontrerons au début), il existe des algorithmes efficaces nous donnant la solution exacte. Dans ces cas nous tentons d'obtenir la solution exacte.

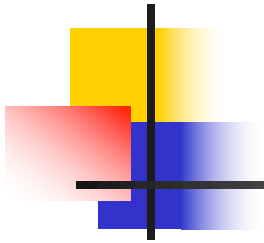
- Cependant:
  - La majorité des problèmes numériques ont comme particularité que la solution exacte ne peut pas être obtenue en un temps fini. Dans ces cas, on se contente d'une solution approximative.
    - Ex: Il est impossible d'obtenir la valeur exacte d'un nombre irrationnel. Cependant, nous pouvons généralement obtenir les 100 premières décimales assez rapidement.
  - Les problèmes NP-complets ont la caractéristique qu'il n'existe (probablement) pas d'algorithme efficace (voir chap 10). Cependant il existe souvent des algorithmes efficaces qui nous donnent une solution très près de l'optimal (voir chap 11). Dans ces cas, on se contente d'une solution approximative.



Choisir les structures de données appropriées

- Certains problèmes ne requièrent rien d'astucieux à cet égard
  - C'est le cas ici pour notre problème de trouver le plus grand commun diviseur entre 2 entiers non négatifs

- Cependant certains algorithmes (voir chap 6 et 7) requièrent une structuration particulière des données (ex: arbre, monceau, graphe…)
  - Revoir les structures de données de «base» à la section 1.4 :
    - Listes chaînées
    - Graphes
    - Arbres
    - Ensembles



Concevoir l'algorithme

- C'est l'étape qui requiert le plus d'ingéniosité et d'expérience
- L'objectif principal de ce cours est de vous apprendre à maîtriser les principales techniques de conception d'algorithmes
  - Chaque technique s'appuie sur une idée simple et qui a fait ses preuves pour de nombreux types de problèmes. Nous illustrerons chaque technique pour solutionner ces types de problèmes:
    - Tri
    - Recherche («search»)
    - Manipulation de chaînes («string processing»)
    - Exploration de graphes
    - Problèmes combinatoires
    - Problèmes géométriques
    - Problèmes numériques
- Tentons de concevoir un algorithme pour trouver le plus grand commun diviseur (pgcd) entre 2 entiers non négatifs

L'approche « force brute » pour le pgcd(m,n)

- On se base ici uniquement sur l'énoncé du problème et la définition du pgcd(m,n)
  - On exploite aucune autre propriété
- s.p.d.g.: m ≥ n
- On tente alors de diviser m et n par t (initialisé à n).
  - S'il y a un reste on essaie de nouveau avec t ← t -1
  - … jusqu'à ce que t=1.

- Pour décrire plus précisément un algorithme, nous utilisons un pseudo-code
  - Voici le pseudo-code de cet algorithme:

ALGORITHME pgcdFB(m,n)
//Trouve le pgcd(m,n) par force brute
//Entrée: Deux entiers non négatifs m ≥ n
// m > 0
//Sortie: pgcd(m,n)

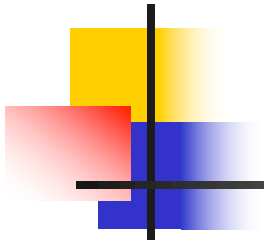
if n=0 **return** m
else
  t ←n
  while (m mod t ≠ 0 **or** n mod t ≠ 0)
    t ← t-1
  return t

- Remarque: m mod t signifie le reste de m/t . Donc (m mod t) ∈ {0..t-1}.

Prouver que l'algorithme est correct

- Un algorithme est correct si et seulement s'il retourne la solution en un temps fini pour toutes les instances possibles du problème.
  - Il faut donc effectuer une preuve rigoureuse
- Pour notre exemple: **L'algorithme pgcdFB est correct.**
- Preuve:
  - Si n=0: alors pgcd(m,n) = m et l'algorithme pgcdFB(m,n) retourne m comme il se doit.
  - Le prédicat testé par **while** est **faux** si et seulement si t est un diviseur (sans reste) de m **et** de n.
  - Si n>0: le prédicat est testé au plus n fois, car ce prédicat est nécessairement **faux** lorsque t=1.
  - Puisque t décroît de 1 à chaque fois que le prédicat est **vrai** et que t débute avec la valeur maximale n, la première valeur de t ne satisfaisant pas le prédicat (et donc la valeur retournée par l'algorithme) sera égale au pgcd(m,n). CQFD

- L'algorithme serait incorrect sans la présence de l'étape **if** n=0 **return** m, car m mod 0 n'est pas défini et pgcdFB échouerait lorsque n=0.



Analyser l'algorithme pour déterminer son efficacité

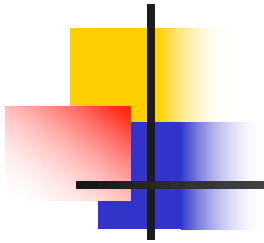
- Un algorithme est efficace lorsqu'il consomme peu de ressources
- Nous portons notre attention principalement sur deux ressources:
  - L'espace mémoire utilisé par l'algorithme
  - Le temps requis par l'algorithme pour fournir la solution

- Il est habituellement (mais pas toujours) assez simple de déterminer l'espace mémoire maximale utilisée par un algorithme.
  - Ex: l'espace utilisé par pgcdFB est simplement l'espace requis par le code et les 3 variables (m,n et t) pour toutes les instances possibles.

- Notre attention portera surtout sur l'efficacité temporelle, car elle est (généralement) plus difficile à déterminer et (généralement) plus déterminante de la performance de l'algorithme.

Trois principales méthodes d'analyse

- L'analyse du pire cas consiste à identifier l'instance(s) sur lequel le temps d'exécution de l'algorithme sera le plus long possible. Désignons ce temps par T_{worst}.
- L'analyse du meilleur cas consiste à identifier l'instance(s) sur lequel le temps d'exécution de l'algorithme sera le plus court possible. Désignons ce temps par T_{best}.
  - Alors $T_{best} \leq T \leq T_{worst}$ pour le temps d'exécution T de n'importe quelle instance.
- L'analyse en cas moyen consiste à calculer le temps moyen d'exécution, c'est-à-dire: calculer la moyenne des temps d'exécution sur toutes les instances possibles.
  - Dans ce cas il faut fournir une distribution de probabilité réaliste sur l'ensemble des instances possibles (voir exemples plus tard).
- Les temps d'exécution seront exprimés à l'aide de la notation asymptotique qui a pour effet de regrouper tous les temps d'exécution qui ne diffèrent que d'une constante multiplicative sous une même catégorie (voir chap 2).



Analyse de l'efficacité de l'algorithme pgcdFB

- Pour notre exemple, exprimons simplement le temps d'exécution en fonction du nombre de fois que le prédicat du **while** est testé.
  - Nous verrons au prochain chapitre que ce temps d'exécution diffère du temps réel seulement par une constante multiplicative

- **Les meilleurs cas** sont ceux ayant n=0. Pour tous ces cas, nous avons T = 0. Alors $T_{best} = 0$.

- **Les pires cas** sont ceux pour lesquels pgcd(m,n) = 1. Ceci se produit, par exemple, lorsque n = m-1. (Pourquoi?) Dans tous ces cas nous avons T = n. Alors $T_{worst} = n$.

Analyse de l'efficacité de l'algorithme pgcdFB (suite)

- L'algorithme pgcdFB est-il efficace?

- Supposons que nous nous limitons aux instances tels que m et n soient des entiers non signés occupant au plus 64 bits
  - Dans ce cas: $T_{worst} = n_{max} = m_{max}\text{-}1 = (2^{64} - 1) - 1 > 2^{63}$
  - Puisque $2^{10} > 10^3$ (c'est-à-dire 1024 > 1000) on a:

    $T_{worst} > 2^{63} = 2^3 \times 2^{60} = 8 \times (2^{10})^6 > 8 \times (10^3)^6 = 8 \times 10^{18}$
  - Un ordinateur pouvant exécuter 10^9 fois par seconde la boucle **while**, exécutera pgcdFB, dans le pire cas, durant plus de 8×10^9 secondes. Ce qui excède 250 années!!

  - C'est trop lent!

Analyse de l'efficacité de l'algorithme pgcdFB (suite)

- L'algorithme pgcdFB s'exécute en un temps linéaire en n (dans le pire cas). Cela signifie que son temps d'exécution augmente exponentiellement en fonction le la taille de n.

- En effet, la taille S(n) d'un entier n > 0 est le nombre de bits utilisés pour représenter n
  - Alors: $S(n) = \lceil \log_2(n+1) \rceil \leq \log_2 n + 1$ (voir rappels page suivante)
  - Alors: $T_{worst} = n \geq 2^{S(n)-1}$

- L'algorithme pgcdFB est donc inefficace, car, dans le pire cas, son temps d'exécution augmente exponentiellement rapidement en fonction de la taille de l'instance du problème.
  - Ici la taille de l'instance est $\lceil \log_2(m+1) \rceil + \lceil \log_2(n+1) \rceil$

- Tentons de concevoir un algorithme efficace pour trouver pgcd(m,n)

Rappel mathématique: logarithmes

- Définition: $x = \log_a(y)$ si et seulement si $a^x = y$

- Notations:
  - log(x) dénote $\log_a(x)$ pour une base quelconque non spécifiée
  - ln(x) dénote $\log_e(x)$
  - lg(x) dénote $\log_2(x)$

- Propriétés (voir exercices série 0):
  - $\log(xy) = \log(x) + \log(y)$
  - $\log(x/y) = \log(x) - \log(y)$
  - $\log(x^y) = y \log(x)$
  - $\log_a(x) = \log_a(b) \log_b(x)$
    - Preuve: $y = \log_b(x)$ ssi $b^y = x$ ssi $y \log_a(b) = \log_a(x)$ CQFD.

Parenthèse mathématique: partie entière

- Soit $x \in R$
- $\lfloor x \rfloor$ dénote la partie entière de x: le plus grand entier $\leq x$
- $\lceil x \rceil$ dénote la partie entière supérieure de x: le plus petit entier $\geq x$

- Propriétés:
- $\forall x \in R$ on a: $x-1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x+1$
- $\forall x \in R$ et $\forall n \in N$ on a: $\lfloor x+n \rfloor = \lfloor x \rfloor + n$ et $\lceil x+n \rceil = \lceil x \rceil + n$
- $\forall n \in N$ on a: $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$
- $\forall n \in N^+$ on a: $\lceil \lg(n+1) \rceil = \lfloor \lg(n) \rfloor + 1$ (notation: $N^+ = N \setminus \{0\}$)
  - voir page suivante pour la preuve du dernier énoncé

Rappel sur la taille d'un entier

- Nous utilisons la représentation binaire usuelle d'un entier non signé
- Soit b le nombre de bits utilisés pour représenter un entier de valeur n
- Nous avons:

$$n = \sum_{i=0}^{b-1} a_i 2^i \quad \text{avec } a_i \in \{0, 1\}$$

- Le plus petit entier utilisant b bits est 1000…0 = 2^{b-1}
- Le plus grand entier utilisant b bits est 1111…1 = $2^b - 1$

- Alors : $2^{b-1} \leq n < 2^b \Rightarrow b - 1 \leq \lg(n) < b \Rightarrow \lfloor \lg(n) \rfloor = b - 1$
- On a aussi: $2^{b-1} < n+1 \leq 2^b \Rightarrow b - 1 < \lg(n+1) \leq b \Rightarrow \lceil \lg(n+1) \rceil = b$

- Donc la taille S(n) d'un entier n est donnée par $b = \lfloor \lg(n) \rfloor + 1 = \lceil \lg(n+1) \rceil$

De retour à l'étape de conception

- Nous verrons que la technique diminuer pour régner consiste à exploiter la relation entre une solution à une instance et une solution à une autre instance plus petite
  - Or il existe justement une telle relation ici
- **Théorème**: pgcd(m,n) = pgcd(n, m mod n)
  - **Démonstration**: Démontrons d'abord que si d divise deux entiers u et v alors d divise également u + v et u – v.
    - En effet, dans ce cas, $\exists s, t \in N$: u = sd et v = td. Alors: $\mathbf{u \pm v = (s \pm t)d}$
    - De plus: d divise u ⇒ d divise un multiple entier ku de u, car ku = ksd
    - Alors: d divise m et n ⇒ d divise m – qn = m mod n
    - De plus: d divise n et m mod n = m – qn) d divise (m-qn) + qn = m.
    - Alors (m,n) et (n, m mod n) ont le même ensemble de communs diviseurs, incluant le plus grand élément de l'ensemble = pgcd. **CQFD**.
- Ainsi, nous pouvons répéter cette égalité jusqu'à ce que m mod n = 0.
  - Exemple: pgcd(60,24) = pgcd(24,12) = pgcd(12,0) = 12

Démonstration alternative

- **Théorème**: pgcd(*m*,*n*) = pgcd(*n*, *m* mod *n*)
- **Démonstration:**
  - Soit *d* un diviseur de *m* et *n*.
  - Il existe donc deux entiers *s* et *t* tels que *m* = *sd* et *n* = *td*.
  - Voici la définition du modulo.

$$m \bmod n = m - \left\lfloor \frac{m}{n} \right\rfloor n$$

  - En remplaçant *m* et *n* par *sd* et *td*, nous démontrons que *d* est également un diviseur de *m* mod *n*.

$$\begin{aligned} m \bmod n &= m - \left\lfloor \frac{m}{n} \right\rfloor n \\ &= sd - \left\lfloor \frac{m}{n} \right\rfloor td \\ &= d\left(s - \left\lfloor \frac{m}{n} \right\rfloor t\right) \end{aligned}$$

Remarque: Le nombre entre parenthèses est un entier, donc *d* est un diviseur de *m* mod *n*.

Preuve alternative (suite)

- **Démonstration (suite):**
  - Soit *e* un diviseur commun à *m* mod *n* et *n*.
  - Il existe donc deux entiers *p* et *q* tels que

$$m \bmod n = pe$$

$$n = qe$$

  - En effectuant les manipulations suivantes, nous concluons que *e* est également un diviseur de *m*.

$$m \bmod n = m - \left\lfloor \frac{m}{n} \right\rfloor n$$ Définition du modulo

$$m = m \bmod n + \left\lfloor \frac{m}{n} \right\rfloor n$$ Isolation de *m*

$$m = pe + \left\lfloor \frac{m}{n} \right\rfloor qe$$ Substitution de *m* mod *n* et *n* par *pe* et *qe*

$$m = e\left(p + \left\lfloor \frac{m}{n} \right\rfloor q\right)$$ Mise en évidence de *e*. Le nombre entre parenthèses est un entier, donc *e* est un diviseur de m.

Preuve alternative (conclusion)

- **Démonstration (suite):**
  - Nous avons prouvé qu'un diviseur commun à *m* et *n* est également un diviseur de *m* mod *n*.
  - Nous avons prouvé qu'un diviseur commun à *m* mod *n* et *n* est également un diviseur de *m*.
  - Par conséquent, les diviseurs communs à (*m*, *n*) sont les mêmes diviseurs communs à (*n*, *m* mod *n*), y compris le plus grand.
  - PGCD(*m*, *n*) = PGCD(*n*, *m* mod n). **C.Q.F.D.**

- Ainsi, nous pouvons répéter cette égalité jusqu'à ce que m mod n = 0.
  - Exemple: pgcd(60,24) = pgcd(24,12) = pgcd(12,0) = 12

L'algorithme d'Euclide

- Cette réduction nous donne l'algorithme suivant (proposé par Euclide de la Grèce antique):

ALGORITHME Euclide(m,n)
//Entrée: Deux entiers non négatifs m ≥ n
// et m > 0
//Sortie: pgcd(m,n)

while n > 0 **do**
  r ← m mod n
  m ← n
  n ← r
return m

- Cet algorithme est correct en vertu du théorème précédent **et** du fait que n diminue toujours à chaque itération (car m mod n < n).

- Cet algorithme est-il plus efficace que pgcdFB ?
  - Oui, car la taille de l'instance diminue « habituellement » plus rapidement et jamais plus lentement.

- Cet algorithme est-il significativement plus efficace?

- Son temps d'exécution en pire cas, T_{worst}, est-il exponentiel ou polynomial? (en fonction de la taille de l'instance)
  - Seule une analyse rigoureuse en pire cas peut nous fournir la réponse…

Analyse en pire cas de l'algorithme d'Euclide

- Donc n diminue **au moins** d'une unité après chaque itération de la boucle.
  - Ce fait, à lui seul, nous donne encore $T_{worst} = n$: un algorithme exponentiel en la taille de l'instance. Progressons-nous davantage?

- Un élément de la réponse nous est donné par le résultat suivant:
  - **$\forall$ m $\geq$ n on a toujours m mod n < m/2**
  - **Preuve**:
    - Si n > m/2 : alors $1 \leq m/n < 2 \Rightarrow \lfloor m/n \rfloor = 1$
      $\Rightarrow$ m mod n = $m - n\lfloor m/n \rfloor = m - n < m - m/2 = m/2$.
      Alors m mod n < m/2 dans ce cas.
    - Si n $\leq$ m/2 : alors m mod n < m/2 (dans ce cas également).
      CQFD.

Analyse en pire cas de l'algorithme d'Euclide (suite)

- À chaque itération la boucle assigne alors à n une valeur < m/2 et assigne à m l'ancienne valeur de n.
- Alors m diminue toujours au moins de moitié après deux itérations successives!
- Après une paire d'itérations: la nouvelle valeur m' < m/2
- Après k paires d'itérations: $m' < m \times 2^{-k}$
- Alors, le nombre de paires d'itérations ne peut pas excéder k^* : le plus petit entier k satisfaisant $m \times 2^{-k} < 1$ (car dans ce cas m' doit être nul).
- Alors $k^* = \lceil \lg(m) \rceil$.
- Dans le pire des cas, Euclide(m,n) effectue alors **au plus** $2\lceil \lg(m) \rceil$ itérations. Alors $T_{worst} \leq 2\lceil \lg(m) \rceil$
- Puisque la taille S de l'instance est $\lceil \lg(m+1) \rceil + \lceil \lg(n+1) \rceil > \lceil \lg(m) \rceil$, on a $T_{worst} < 2S$. **Alors le temps d'exécution de l'algorithme d'Euclide croit (au plus) linéairement avec la taille de l'instance!!**

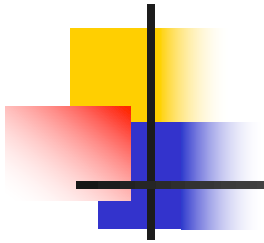
Conséquences pratiques

- Sur des entiers pouvant être codés sur 64 bits, l'algorithme d'Euclide effectuera au plus $2 \times \lceil \lg(2^{64}) \rceil = 128$ itérations (dans le pire des cas).
  - Sur un ordinateur conventionnel « ordinaire », le résultat sera obtenu en moins de 10^{-6} secondes.

- Or, dans le pire des cas, pgcdFB nécessitera au moins 2^{63} itérations!
  - Sur un ordinateur conventionnel « ordinaire », le résultat sera obtenu en plus de 250 années! (dans le pire des cas)

- (dans le meilleur des cas, chaque algorithme nécessite 0 itération)

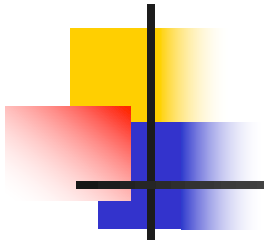
- Quel algorithme choisiriez-vous?



Deux leçons importantes à retenir

- En s'inspirant d'une technique connue et éprouvée de conception d'algorithme (ici, diminuer pour régner), nous avons pu concevoir un algorithme exponentiellement plus efficace qu'un algorithme conçu par une approche de force brute.

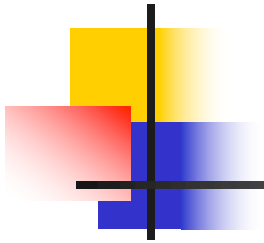
- En analysant rigoureusement le temps d'exécution de ce nouvel algorithme en pire cas, nous avons pu fournir une certification précise sur sa très grande efficacité.



Les deux principaux objectifs du cours

- Vous enseigner les principales techniques de conception d'algorithmes qui ont fait leurs preuves dans plusieurs domaines et les appliquer à des problèmes réels en informatique.
  - Ceci augmentera votre compétence à concevoir des algorithmes efficaces

- Vous enseigner les principales techniques d'analyse d'algorithme et les appliquer à des problèmes réels en informatique.
  - Ceci augmentera votre compétence à fournir une certification de l'efficacité de votre nouvel algorithme



Coder l'algorithme

- Si nous sommes satisfaits de l'efficacité de notre algorithme, il faudra écrire l'algorithme dans un langage de programmation
  - et en produire un exécutable
- Un programmeur expérimenté peut produire un code substantiellement plus efficace que celui produit par un novice
- Mais l'efficacité du code ne différera que d'une constante multiplicative lorsque l'algorithme est le même
  - L'amélioration est très significative lorsqu'il s'agit d'une constante élevée (exemple: 10)
- Puisque le choix de l'algorithme affecte l'ordre de croissance (ex: exponentiel versus linéaire) du temps d'exécution, le choix de l'algorithme a plus d'impact sur le temps d'exécution que l'optimisation du code.
- Il faudra écrire votre programme d'une manière à ce qu'il soit facile à maintenir par autrui: ceci fait l'objet d'un autre cours.
- Il faudra tester et valider votre programme: ceci fait l'objet d'un autre cours