Bases d'algorithmique et de programmation

Stage Liesse

François Pessaux

`https://perso.ensta-paris.fr/~pessaux/teaching/liesse`

Laboratoire U2IS
ENSTA Paris
2021

ENSTA

IP PARIS

Bienvenue à toutes et tous. . .

- Le sujet :

  transmettre les bases de l'algorithmique et de la programmation
- Le présentateur :
  - Enseignant-chercheur en informatique à l'ENSTA depuis +9 ans.
  - Responsable de l'informatique en $1^{\text{ère}}$ année.
  - Cours en 1A et 2A.
  - Responsable de parcours de 3A.
- Le plan :
  - Connaissances et compétences actuelles des élèves.
  - Points généraux considérés importants.
  - Sensibilisation à « l'efficacité » (complexité).
  - Paradigmes généraux de programmation.
  - Exemple d'animation de travaux pratiques.

Panorama actuel

Constat côté enseignants

- Problèmes déjà traités principalement mathématiques.
- Solution mathématique compliquée mais informatique triviale.
- Recours courants à des bibliothèques masquant les difficultés algorithmiques.

- Manque d'esprit d'abstraction.
- Difficultés à décomposer un problème en sous-problèmes.
- Pas de préoccupation d'efficacité ni de clarté des programmes.
- Solutions guidées par les constructions du langage connu.

Objectifs d'un cours de bases d'algo. et prog.

- Apprendre à concevoir des algorithmes.
- Apprendre à se poser des questions sur la spécification.
- Apprendre à penser « algorithmiquement ».
- Apprendre à penser indépendamment d'un langage.
- Acquérir une démarche systématique et scientifique :
  1. d'analyse,
  2. de formalisation,
  3. de conception,
  4. d'implantation.
- ~~Faire des spécialistes~~.
- ~~Étudier des algorithmiques « bien connus »~~.
- ~~Étudier des structures de données « bien connues »~~.

Concevoir des algorithmes

Algorithme

- ~~Une recette de cuisine~~.
- ~~Un langage de programmation~~
  - Langage = juste des mots pour décrire un algorithme.
- Méthode de passage d'un ensemble de données D à un ensemble de résultats R.
- Méthode s'appuyant sur du calcul.
- ⇒ Fonction (qui peut être partielle).

LA première question...

« *Qu'est-ce que j'ai, qu'est-ce que je veux ?* »

1. Les données à traiter : « entrées », hypothèses du problème.
2. Les résultats à obtenir : « sorties ».

Exemple : « *Je veux calculer une racine carrée.* »

- En entrée j'ai un « nombre » x.
- En sortie j'ai un « nombre » y tel que $y^2 = x$ ... ou pas.

C'est après que l'on se demande comment le faire.

Liens entre entrées et sorties

Liens entrées → sorties :

1. Celui exprimé par le futur utilisateur : spécification.
2. Celui décrit par le calcul : algorithme puis programme.

Un algorithme (puis un programme) « fonctionne bien » ssi (2) satisfait (1).

Spécification

Spécification

- Spécification : expression du besoin.
- ∃ plusieurs formes de spécifications :
  - Simple formule mathématique : expression directement implantable dans un langage.
  - Formule mathématique plus complexe nécessitant une mise en forme « informatique ».
  - Problème décrit en langage naturel et informel.
- +/- besoin de modéliser et raffiner.

Spécification simple

« Écrire un programme permettant de convertir une température en degrés Fahrenheit vers des degrés Celcius »

- x °F $\longrightarrow \frac{5}{9}(x-32)$ °C
- Expression arithmétique ⇒ direct dans le langage.

```
float ftoc (float t) {
  return (5.0 / 9.0 * (t - 32.0)) ;
}
```

- Traduction simple.

Spécification intermédiaire

« *Écrire un programme permettant de calculer f(n) sachant que :*

$$\begin{cases} f(0) &= 1 \\ f(n) &= \prod_{k=1}^{n} k \end{cases}$$

- Deux cas dans la définition
  - Faire explicitement 2 cas ?
  - Possibilité d'un seul cas général ?
- Répétition : introduction d'un mécanisme d'itération
  - Faire une boucle ?
  - Faire de la récursion ?

```
int f (int x) {
  int res = 1 ;
  while (x > 1) {
    res = res * x ;
    x = x - 1 ;
  }
  return res ;
}
```

```
int f (int x) {
  if (x <= 0) return 1 ;
  return (x * f (x - 1)) ;
}
```

Spécification hautement non triviale

« Écrire un programme permettant de calculer les déformations d'une structure »

« Écrire un programme permettant de naviguer sur le WEB »

« Écrire un programme permettant de simuler les cours de la bourse »

- Modéliser le problème.
- Décomposer en sous-problèmes.

Domaines des entrées et des sorties

Importance des domaines (1/2)

« Écrire un programme qui calcule la racine carrée d'un nombre donné en entrée. »

- Domaine de définition : $\mathbb{R}^+$.
- Type informatique `float` : toujours signé.
  - 2 cas : < 0 et $\geqslant 0$ $\Rightarrow$ condition à tester.
- $\Rightarrow$ méthode itérative de Newton...

```
#define X0 (3.0)    /* Random seed > 0. */

float squr (float a, int max_iter) {
  int i ;
  float xn = X0 ;

  if (a < 0) return -1 ; /* Assume -1 is the error value. */
  for (i = 0; i < max_iter; i++)
    xn = 0.5 * (xn + (a / xn)) ;

  return xn ;
}
```

Importance des domaines (2/2)

« Écrire un programme qui calcule la racine carrée d'un nombre donné en entrée. »

- « Nombre » : qui a dit que c'est un réel ?
- Qui a dit que le résultat était un réel ?

- Autre choix : résultat entier ⇒ absence de racine.
- Autre choix : résultat complexe ⇒ aucun cas d'erreur, nécessité de créer des nombres complexes.

- Autres domaines d'entrées / sorties ⇒ algorithme totalement différent.
- ⇒ Nécessité de définir clairement les domaines (entrées et sorties).

Domaine mathématique

- C'est un entier ?
- C'est un entier positif ou nul ?
- C'est un réel ?
- C'est suite finie ordonnée de lettres (mot, chaîne de caractères) ?
- Valeur de vérité (booléen, vrai / faux) ?
- C'est un couple (entier $\times$ booléen) ?
- C'est un ensemble non ordonné de réels ?

- Indique les structures manipulées par l'algorithme.
- Exemples :
  - Date $\Rightarrow$ 3 entiers.
  - Âge $\Rightarrow$ 1 entier.
  - Température $\Rightarrow$ 1 réel ($^\circ$K $\Rightarrow \geqslant 0$).

Domaine sémantique

- C'est une distance ?
  - À échelle astronomique ?
  - À échelle subatomique ?
- C'est une vitesse ?
  - Linéaire ?
  - Angulaire ?
- Dirige les opérations autorisées sur les données.
  - Vitesse + distance = ⛔
  - Vitesse linéaire + vitesse angulaire = ⛔
- Définit les ordres de grandeur, les ϵ.
  - Collision de 2 planètes : distance calculée = 10 m ⇒ oui.
  - Collision de 2 particules : distance calculée = 10 m ⇒ non.

Domaine informatique (type)

- Scalaires informatique = approximation des « nombres » mathématiques.
- Arrondis, finitude, etc.
- Parfois, l'algorithme doit prendre en compte ces différences.

- Exemple : programme de la racine carrée (diapo 16).
- Flottant `float` toujours signés.
- Impossible de restreindre structurellement (par le type de données) à $\mathbb{R}^+$.
- $\Rightarrow$ Un test (`if`) à faire dans l'algorithme.

Conclusion

- Commencer par déterminer les domaines des données :
  1. . . . mathématique,
  2. . . . sémantique,
  3. . . . informatique (type),
- . . . pour entrées et sorties.
- Permet de préciser la spécification (besoin attendu).
  - Que faire dans un cas non (clairement) spécifié ?
- ⇒ Influence l'algorithme (cas admis, cas d'erreur).

Conception par raffinement : exemple

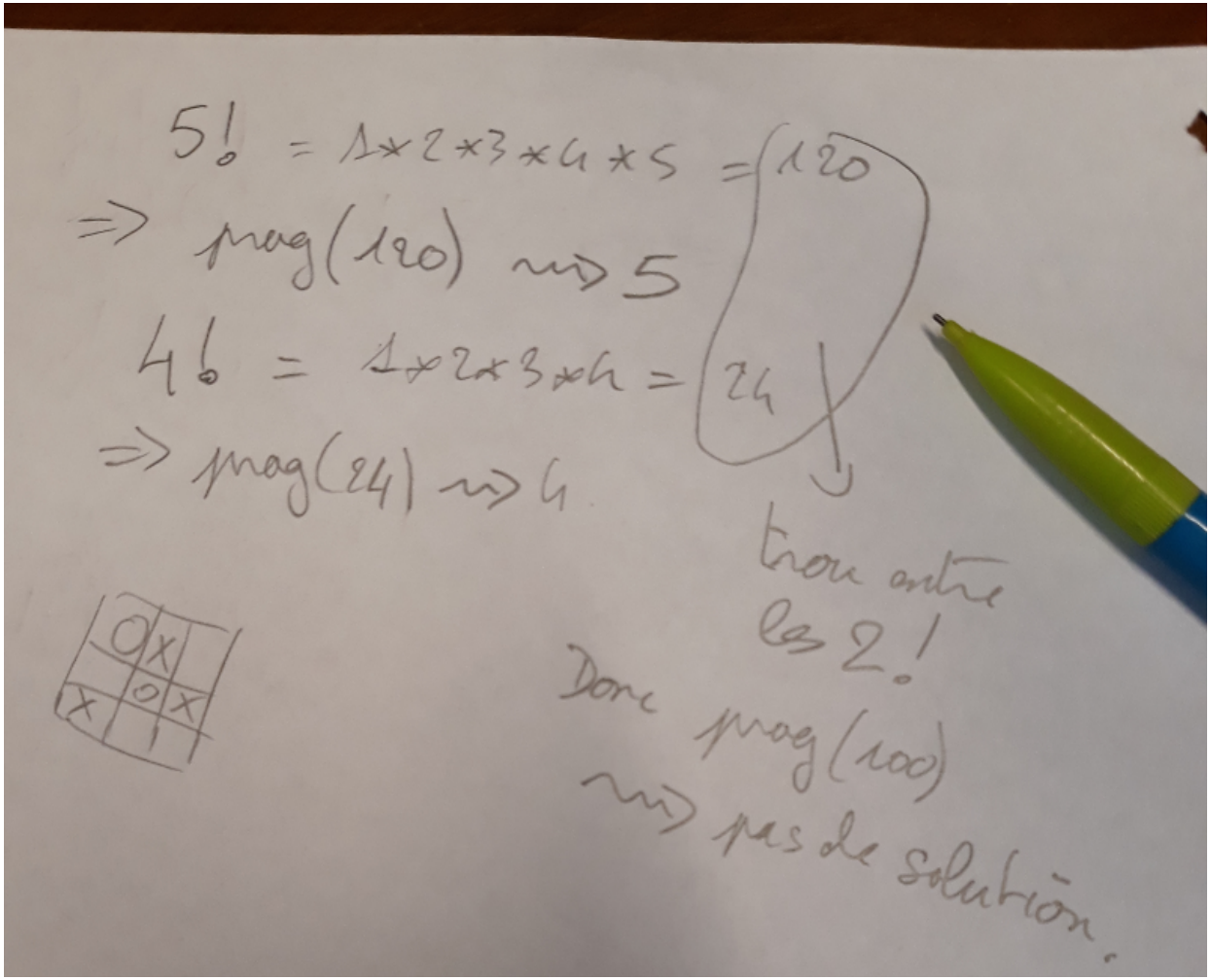
Le sujet

« Écrire un programme qui prend un entier et dit de quoi sa valeur absolue est la factorielle. »

- Formuler le problème plus clairement :
  - « Soit $x \in \mathbb{Z}$, trouver y tel que $y! = |x|$. »
  - C'est en fait la fonction inverse de factorielle qui est demandée.
- Mes entrées : un nombre entier positif, nul ou négatif.
- Mes sorties : un nombre entier positif.

Sûr ?

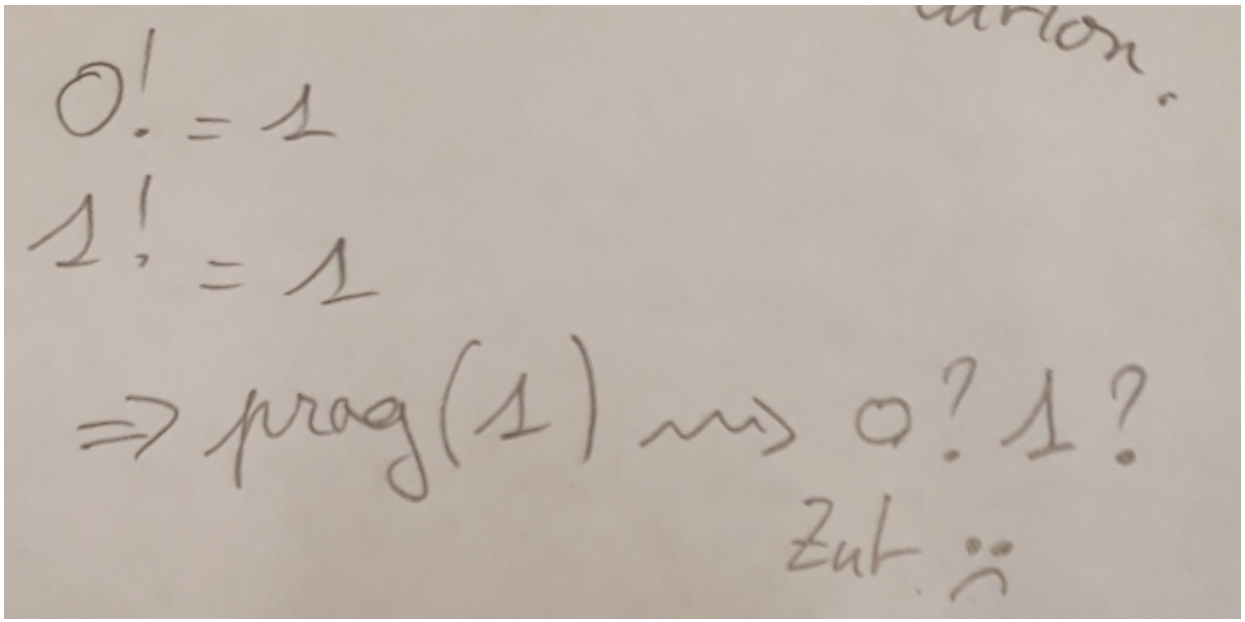
Un peu de papier



- Cas où pas de solution.
- Mes sorties : un nombre entier positif ou une erreur.

Encore un peu de papier

- On refait quelques essais. . .

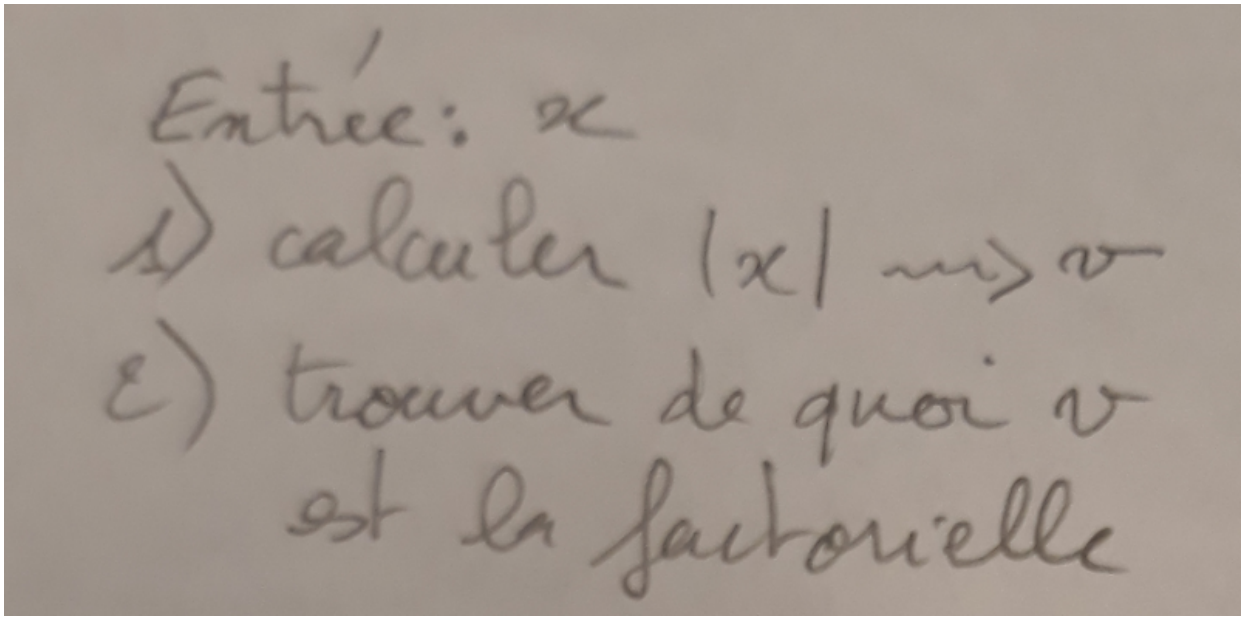


- Même pas une fonction en fait : cas avec 2 résultats possibles.
- Prévoir une autre erreur si entrée = 0, 1 . . . ou -1.

Et maintenant quel algorithme ?

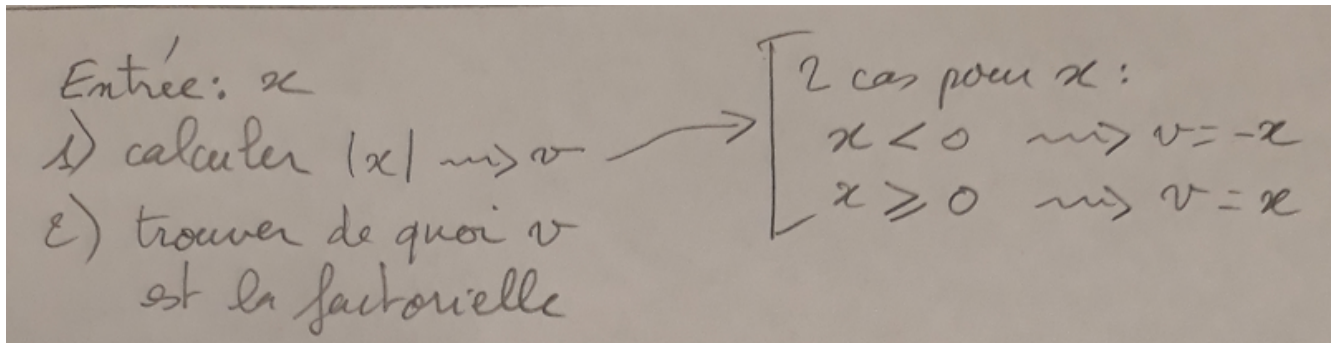
« Écrire un programme qui prend un entier et dit de quoi sa valeur absolue est la factorielle. »

- Découper en sous-problèmes.



Sous-problèmes, sous-sous-problèmes... (1/2)

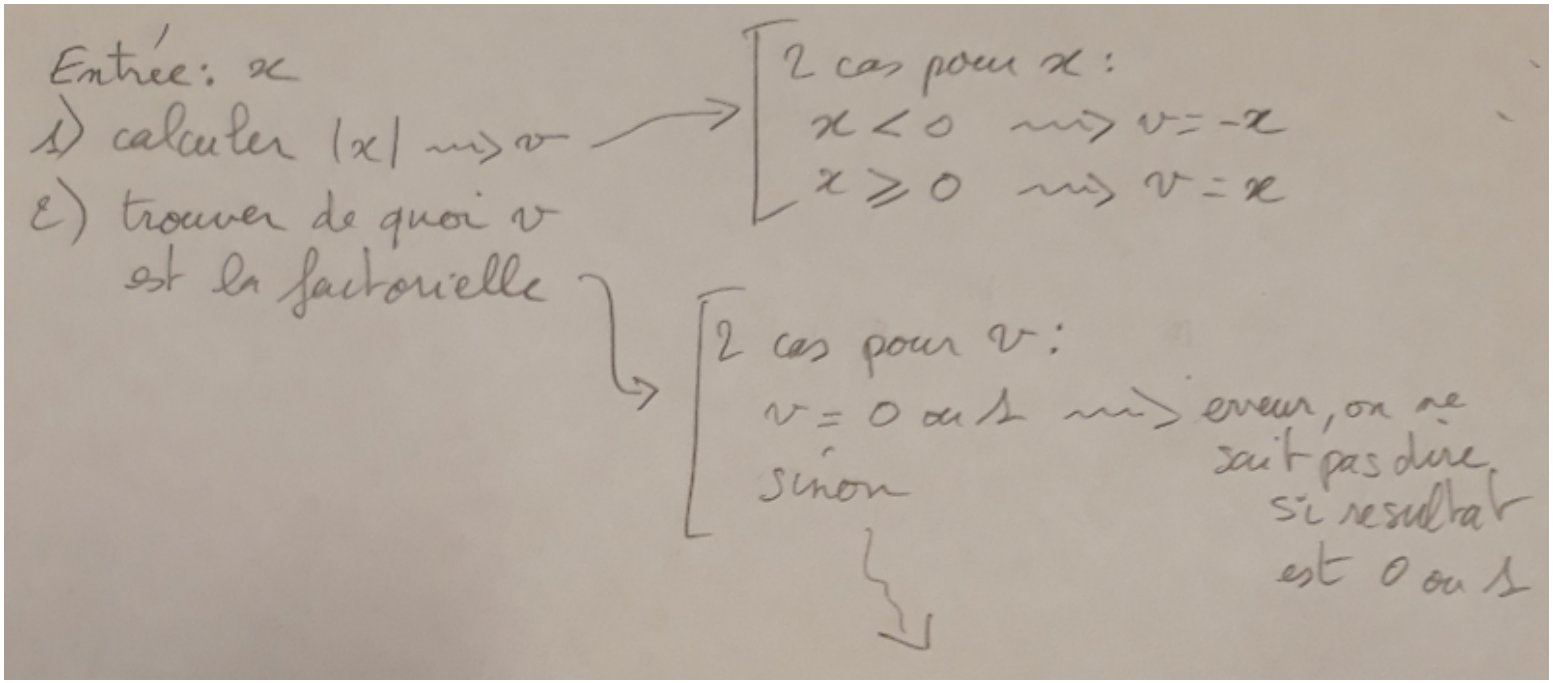
- Problème actuel : calcul de la valeur absolue.
- Deux cas.
  - Lesquels ?
  - Quels résultats pour ce problème ?



- Raisonnement local.

Sous-problèmes, sous-sous-problèmes... (2/2)

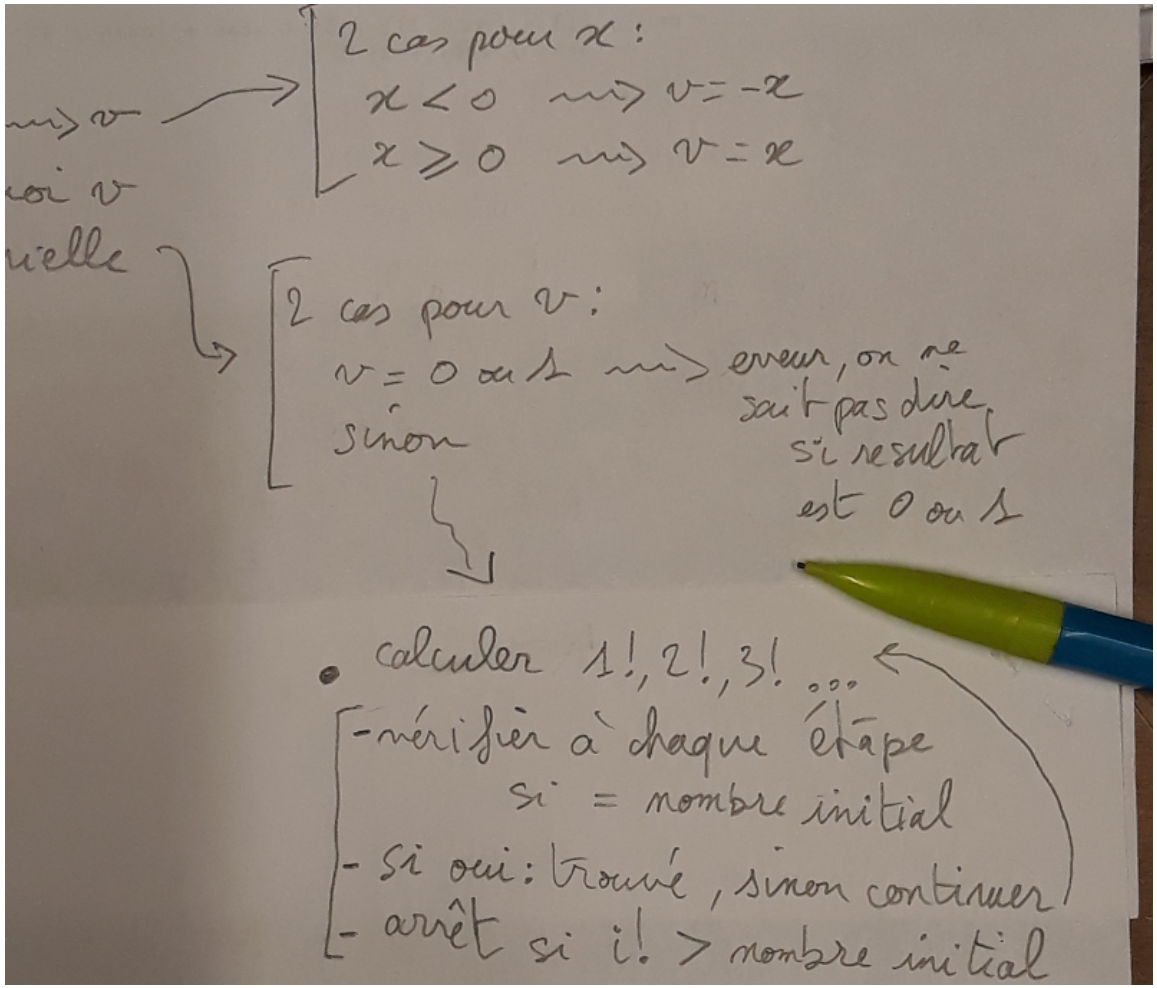
- Problème actuel : calcul de la « factorielle inverse ».
- Deux cas.
  - Lesquels ?
  - Quels résultats pour ce problème ?
    - Utilisation de l'analyse faite en diapo 25.



- Raisonnement toujours local.
- Reste le cas « sinon » à traiter.

Le sous-problème de la factorielle . . . (solution 1)

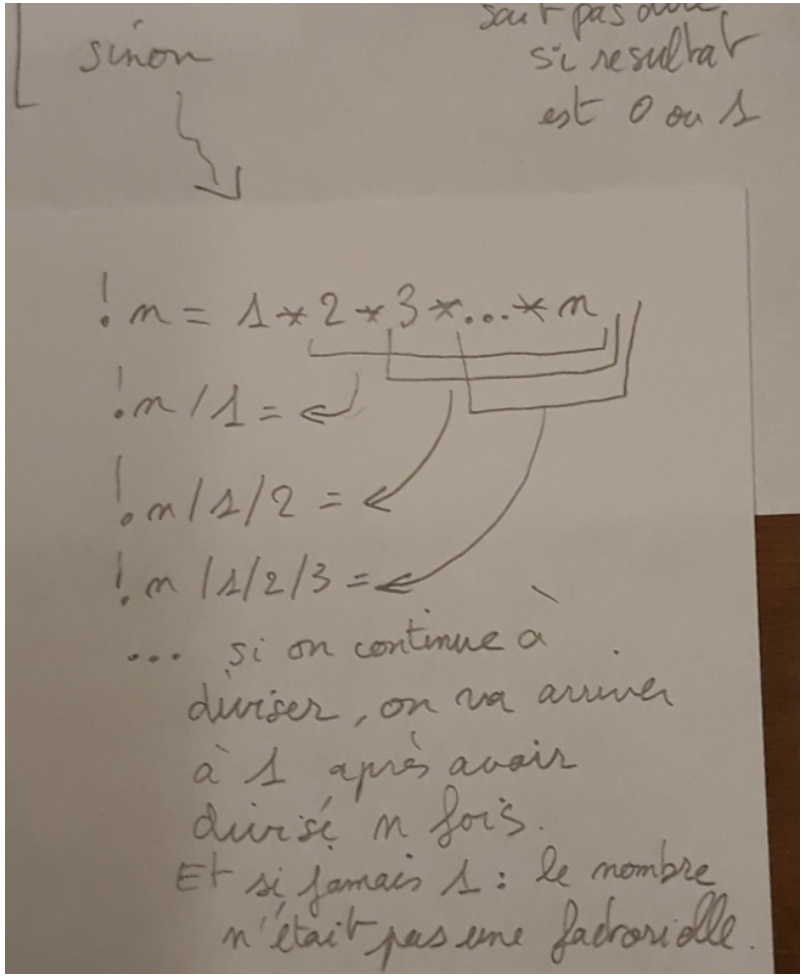
- Comment trouver de quel nombre v est la factorielle ?



- ⇒ Boucle.
- Remarque : inutile de calculer 0! et !1 (déjà gérés).

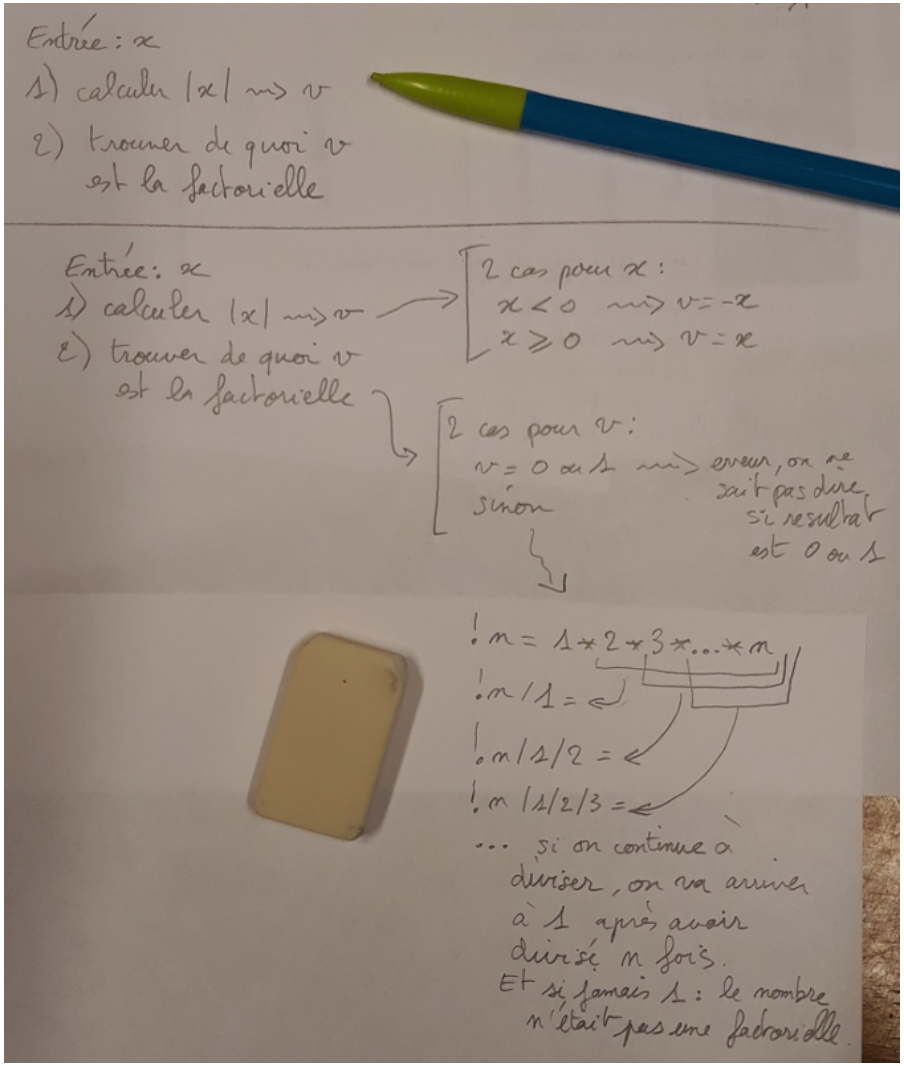
Le sous-problème de la factorielle . . . (solution 2)

- Comment trouver de quel nombre v est la factorielle ?



- ⇒ Boucle.
- Remarque : inutile de diviser par 1.
- Remarque : plus besoin de calculer la factorielle.

L'algorithme complet




```c
#include <stdio.h>
#include <stdlib.h>

int factinv (int x) {
  int i, div, v;
  /* Compute v as |x|. */
  if (x < 0) v = -x;
  else v = x;
  /* Check error. */
  if ((v == 0) || (v == 1))
      return -1;
  div = v;
  for (i = 2; i <= v; i++) {
    div = div / i;
    if (div == 1) return i;
  }
  return -1;
}
```

Constat

- Création de l'algorithme par décomposition en sous-problèmes.
- Raisonnement local à chaque sous-problème.
  - Questions et solutions petit à petit.
- Synthèse des solutions des sous-problèmes imbriqués → solution du problème global.

- Algorithme conçu hors langage de programmation spécifique.
- Langage de programmation : à la toute fin, une fois l'algorithme conçu.

Importance de tester le programme

- Vérifier les résultats : tests.

```
factinv (120) → 5
factinv (−5040) → 7
factinv (0) → −1
factinv (1) → −1
factinv (119) → −1
factinv (−119) → −1
```

- Encore. . .

```
factinv (3) → 2
factinv (−25) → 4
```

Analyse de la défaillance (1/2)

- Tracer le comportement du programme : des `printf`.
- Afficher les variables pertinentes.

```
int factinv (int x) {
  int i, div, v;
  /* Compute v as |x|. */
  if (x < 0) v = -x;
  else v = x;
  /* Check error. */
  if ((v == 0) || (v == 1)) return -1;
  div = v;
  for (i = 2; i <= v; i++) {
    printf ("Before div = %d\n", div);
    printf ("i = %d\n", i);
    div = div / i;
    printf ("After div = %d\n", div);
    if (div == 1) return i;
  }
  return -1;
}
```

Analyse de la défaillance (2/2)

```
----> factinv (3)
Before div =  3
i =  2
After div =  1
2
```

- Lien entre `div` et `i` : `div = div / i`
- Division entière.
- Erreur dans notre modélisation : pas de prise en compte du type des « nombres ».
- Vocabulaire « parlé » imprécis.
  - En mathématiques : $\mathbb{Z} \neq \mathbb{R}$.
  - En informatique : `int` $\neq$ `float`.

Régler son compte au problème

- Solution simple et rapide : utiliser des `float`.
- Domaine de factorielle : $\mathbb{N} \Rightarrow$ pas très élégant d'introduire $\mathbb{R}$.
- Solution « contre nature » apportant d'autres problèmes.

- Réflexion . . .
  « *Quand est-on certain que le calcul réussit lors de la division ?* »
- *Réponse :* « Quand la division tombe juste ».
- « Division tombe juste » $\equiv$ reste $= 0$.
- Opérateur % : reste division entière.

Correction du problème

```
int factinv (int x) {
  int i, div, v ;
  /* Compute v as |x|. */
  if (x < 0) v = -x ;
  else v = x ;
  /* Check error. */
  if ((v == 0) || (v == 1))
      return -1 ;
  div = v ;
  for (i = 2; i <= v; i++) {
    int rem = div \% i ;
    if (rem != 0) return -1 ;
    div = div / i ;
    if (div == 1) return i ;
  }
  return -1 ;
}
```

```
factinv (120) → 5
factinv (-5040) → 7
factinv (0) → -1
factinv (1) → -1
factinv (119) → -1
factinv (-119) → -1
factinv (-25) → -1
factinv (3)  → -1
```

- Moins de tours de boucle effectués.

Conclusion

- Concevoir un algorithme : besoin de rigueur et de détails.
  - Type des données manipulées à exprimer formellement.
  - Étapes de calcul toutes explicitées.
  - Identifier les limites de la solution apportée au problème.
  - Traiter tous les cas possibles dans le périmètre décidé.
- Fait naturellement ressortir la structuration en fonctions :
  - meilleure lisibilité,
  - meilleure réutilisabilité,
  - meilleure maintenabilité.
- Implantation : prendre en compte des aspects techniques du langage de programmation.
  - Après avoir décomposé le problème « suffisamment ».

Complexité

Comparaison

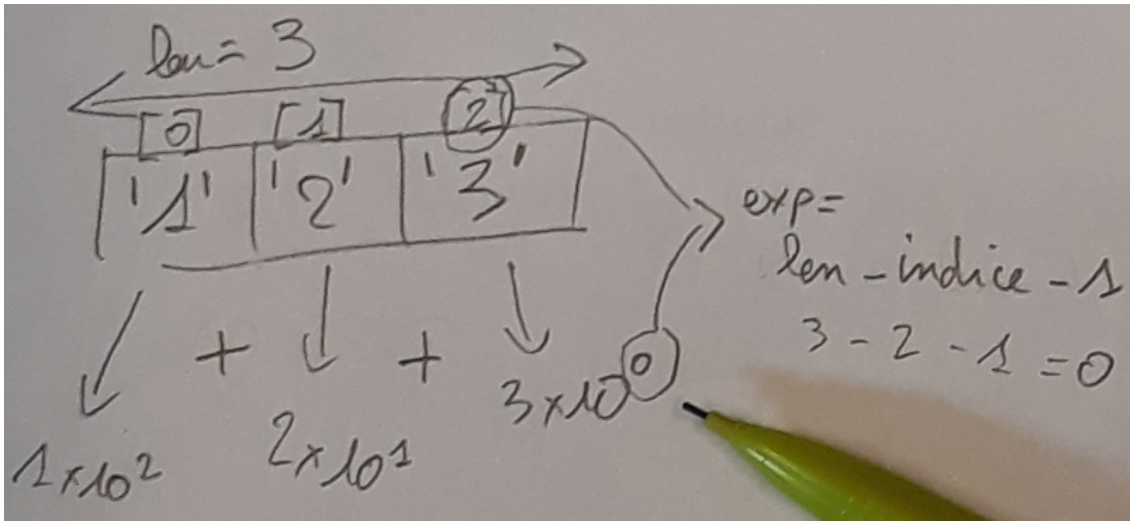
```
int addition (int x, int y) {
  return (x + y) ;
}
```

```
int addition (int x, int y) {
  int res = x ;
  for (int i = 0; i < y; i++)
    res = res + 1 ;
  return res ;
}
```

- L'un est plus court que l'autre.
- Pour x et y donnés :
  - Algorithme de gauche : 1 instruction exécutée.
  - Algorithme de droite : plus de y instructions exécutées.
- Surtout : l'un est plus efficace que l'autre.
- $\nearrow$ instructions $\Rightarrow$ $\nearrow$ temps.
- Efficacité temporelle.

Chaîne vers entier (1/2)

- Conversion chaîne vers entier : `tonum ("123")` → 123.
- Chaîne = tableau de caractères.
- Codes ASCII (entier) des caractères sont ordonnés.



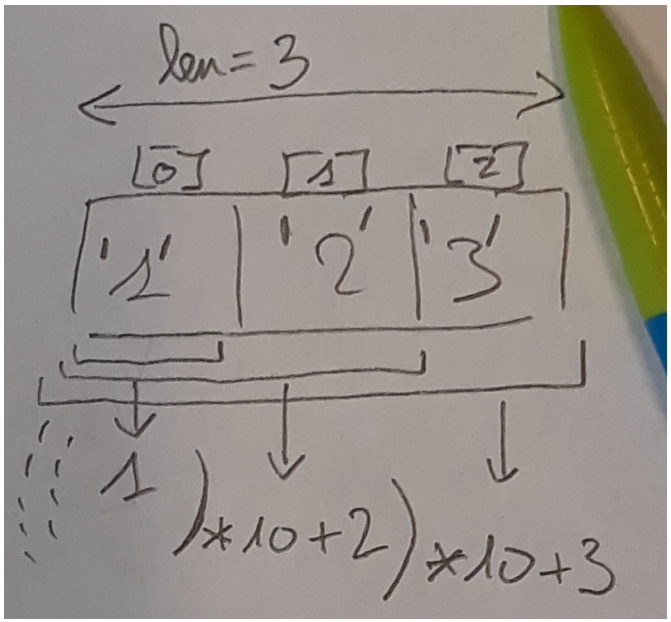

```c
#include <math.h>

/* char = 8 bits integer whose value is the ASCII code of the character. */
unsigned int tonum (char *str) {
  unsigned int res = 0 ;
  int slen = strlen (str) ;

  for (int i = 0; i < slen; i++)
    res = res + (str[i] - '0') * pow (10, (slen - i - 1)) ;
  return res ;
}
```

Chaîne vers entier (2/2)

- Conversion chaîne vers entier : `tonum ("123")` → 123.




```c
/* char = 8 bits integer whose value is the ASCII code of the character. */
unsigned int tonum (char *str) {
  unsigned int res = 0 ;
  for (int i = 0; str[i] != '\0'; i++)
    res = res * 10 + (str[i] - '0') ;
  return res ;
}
```

⇒ RIP pow.

Court vs court

```
unsigned int tonum (char *str) {
  unsigned int res = 0 ;
  int slen = strlen (str) ;
  for (int i = 0; i < slen; i++)
    res = res + (str[i] - '0') *
        pow (10, (slen - i - 1)) ;
  return res ;
}
```

```
unsigned int tonum (char *str) {
  unsigned int res = 0 ;
  for (int i = 0; str[i] != '\0'; i
      ++)
    res = res * 10 + (str[i] - '0') ;
  return res ;
}
```

- Longueurs quasiment identiques.
- Algorithme de gauche : utilisation de `pow` (puissance).
- `pow` est une fonction : plusieurs instructions (boucle en version naïve) .
- `pow` est une fonction sur les flottants.
- Appel pour chaque exposant.
- ⇒ Algorithme de gauche plus coûteux en temps.

Qu'est-ce que la complexité ?

- Complexité d'un algorithme : mesure de son efficacité intrinsèque :
  - en fonction de la taille des données à traiter,
  - asymptotiquement,
  - dans le pire cas ou en moyenne.
- ⇒ Notion d'efficacité indépendante de la vitesse de la machine.
- Deux formes de complexité :
  - Temporelle : s'intéresse au temps passé dans dans l'exécution.
  - Spatiale : s'intéresse à l'espace mémoire nécessaire lors de l'exécution.
- On s'intéresse le plus souvent à la complexité temporelle.
- Pratiquement, pour une entrée de taille n :
  - On compte le nombre d'opérations « *de base* » nécessaires.
  - On regarde comment ce nombre évolue asymptotiquement.

Notations de complexité

- Évaluation de la complexité, de l'efficacité d'un algorithme :
  - $\Rightarrow$ Encadrer le nombre d'opérations qu'il fait.
  - $\Rightarrow$ Borne supérieure et borne inférieure.
  - Lorsque la taille de l'entrée tend vers $+\infty$.
- $O(g)$: ensemble des fonctions f telles que $0 \leq f(x) \leq k \times g(x)$
  - Avec $k \in \mathbb{R}_+^*$
  - Et $\exists x_0 \in \mathbb{N}^*, \forall x \geq x_0$ ($\rightsquigarrow$ comportement asymptotique).
- $\theta(g)$: ensemble des fonctions f telles que $k_1 \times g(x) \leq f(x) \leq k_2 \times g(x)$
  - Avec $k_1, k_2 \in \mathbb{R}_+^*$
  - Et $\exists x_0 \in \mathbb{N}^*, \forall x \geq x_0$ ($\rightsquigarrow$ comportement asymptotique).
- Notation malheureuse : on écrit $f = O(g)$ au lieu de $f \in O(g)$.

Classement de complexités

$$\log(n) \ll \sqrt{n} \ll n \ll n\log(n) \ll n^2 \ll n^3 \ll 2^n \ll exp(n) \ll n! \ll n^n \ll 2^{2^n}$$

- Définitions : complexité
  - linéaire : $f(x) = O(x)$ → réalisable
  - quadratique : $f(x) = O(x^2)$ → réalisable
  - polynomiale : $\exists k > 0,\ f(x) = O(x^k)$ → souvent réalisable
  - exponentielle : $\exists b > 1,\ f(x) = O(b^x)$ → en général irréalisable
  - doublement exponentielle, par exemple : $f(x) = O(2^{2^x})$.
  - sous-exponentielle, par exemple : $f(x) = O(2^{\sqrt{x}})$.
- Quelques exemples :
  - Algorithme de tri par tas : $O(n\ log\ n)$.
  - Calcul de la matrice d'accessibilité d'un graphe : $O(n^3)$.

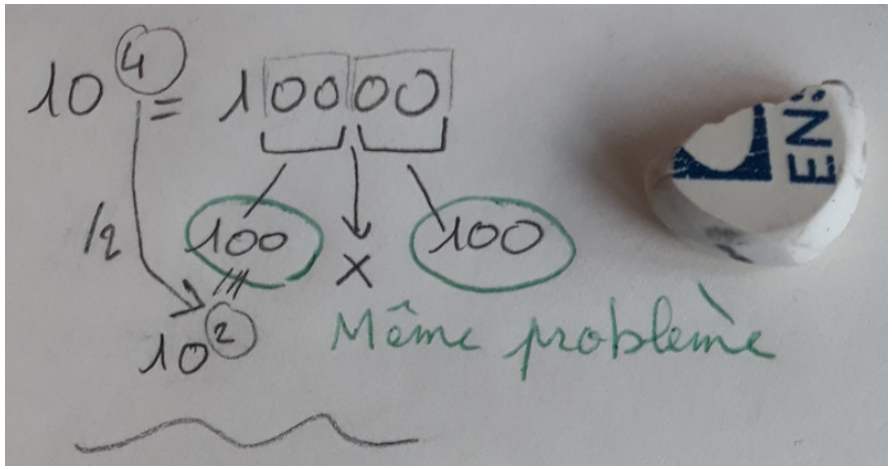
Quelques ordres de grandeur

- (2010) - **AMD FX-8150 (8-core) @ 3.6 GHz** : $\approx 1.0\ 10^{11}$ instr/s.
- (2011) - **Intel Core i7 2600K@ 3.6 GHz** : $\approx 1.2\ 10^{11}$ instr/s.
- (2016) - **Intel Core i7 6950X @ 3 GHz** : $\approx 3.2\ 10^{11}$ instr/s.
- (2017) - **AMD Ryzen 7 1800X@ 3.6 GHz** : $\approx 3\ 10^{11}$ instr/s.

- PC standard, 2^{40} (10^{12}) instructions : pas un problème.
- Records actuels : un peu au-delà de 2^{60} (10^{18}) op. binaires (réalisable par des gens « motivés » → NSA, Folding@home . . .)
- En cryptographie, actuellement 2^{80} (10^{24}) opérations binaires sont considérées comme inatteignables . . . aujourd'hui . . .

Paradigmes de programmation : diviser pour régner

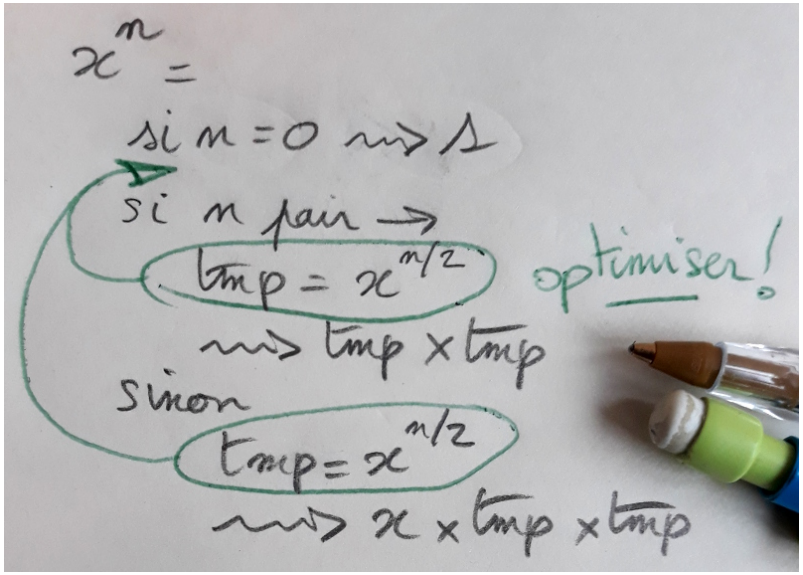
Calculer x^n



- Cas particulier d'utilisation de $x^{a+b} = x^a \times x^b$.
- Sous-problèmes de même forme.
- Essayer de prendre $a = b$ pour ne calculer qu'une fois la puissance.
- Facile : on prend $a = \frac{n}{2}$.

- Et si n impair ?
- Exemple : $10^5 \ldots = 10 \times 10^4$.
- Or 10^4 ($x^{\text{nombre pair}}$), on sait faire efficacement.

x^n : l'algorithme efficace



- Fonctionne car `n` est un entier.
- Si flottant : $1/2 = 0.5$ et non 0 ⇒ boucle infinie.
- Complexité temporelle : $\theta(log(n))$.
- *Quid* complexité spatiale ?

```
int power (int x, int n) {
  if (n == 0) return 1 ;
  int tmp = power (x, n / 2) ;
  if (n % 2 == 0)
    return (tmp * tmp) ;
  else return (x * tmp * tmp) ;
}
```

Paradigme « Diviser pour régner »

- Partitionner le problème d'origine en sous-problèmes de même forme
  - voire identiques
  - $\Rightarrow$ 1 gros problème $\rightsquigarrow$ 1 ou plusieurs problèmes plus petits.
- Résoudre les sous-problèmes (récursivement).
- Recombiner les solutions des sous-problèmes $\rightsquigarrow$ solution du problème initial.

Paradigmes de programmation : programmation dynamique

Le problème à résoudre

- Barre de longueur fixe à couper pour revente.
- Prix de revente connus pour chaque longueur.
- But : maximiser le gain.

Longueur	0	1	2	3	4	5	6	7	8
Prix	0	2	3	8	10	13	15	16	21

- Une solution optimale : bout de 3 + bout de 5 → prix : 21.
- Autre solution optimale : bout de 8 → prix : 21.

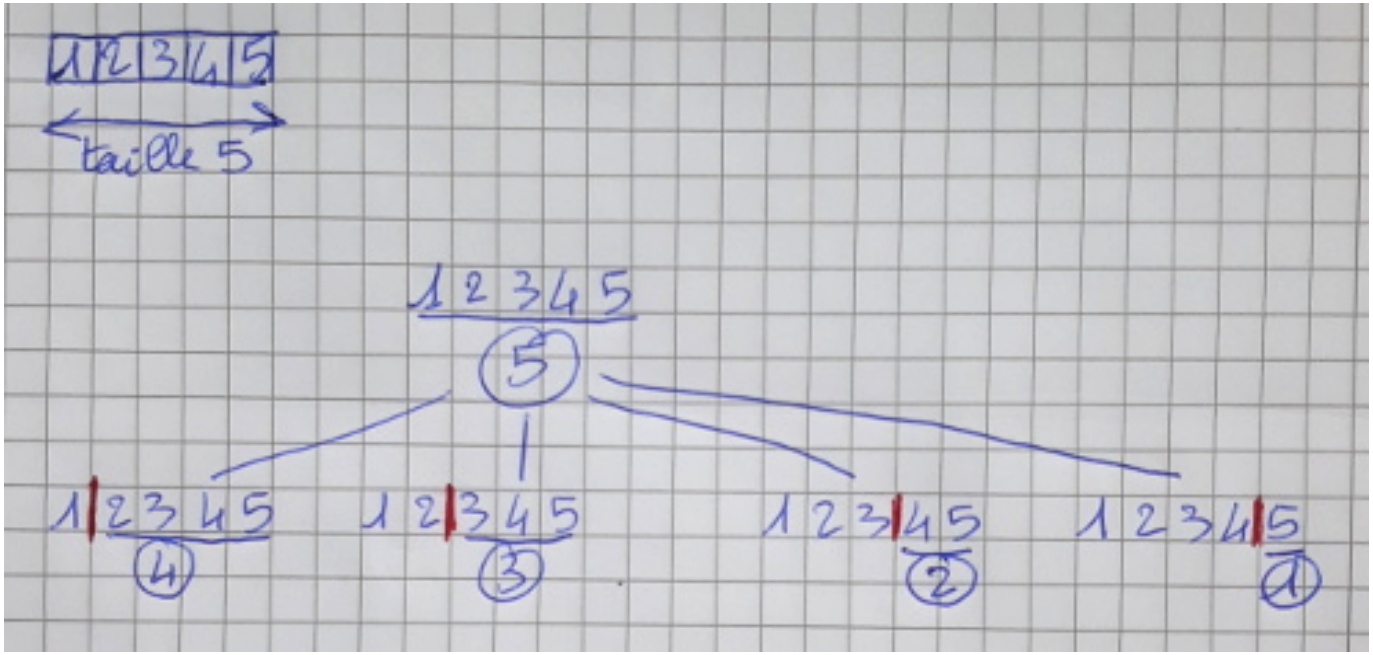
Trouver un algorithme

- Trouver toutes les découpes possibles.
- Calculer pour chacune le gain.
- Retourner le plus grand.

⇒ Comment calculer toutes les découpes possibles ?

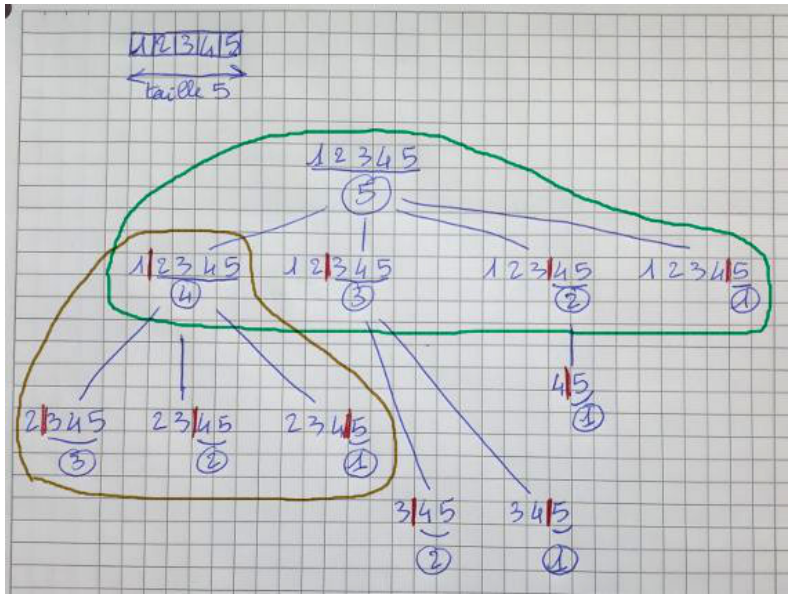
- Commencer par le cas de la barre entière.
- Puis, faire 1 découpe...
- ...faire ça pour toutes les positions possibles de la découpe.
- Recommencer sur de qui reste de chacune des découpes.

Exploration



- Taille $n \Rightarrow n-1$ positions où découper.
- Prochaine étape, recommencer sur le reste de chaque découpe.

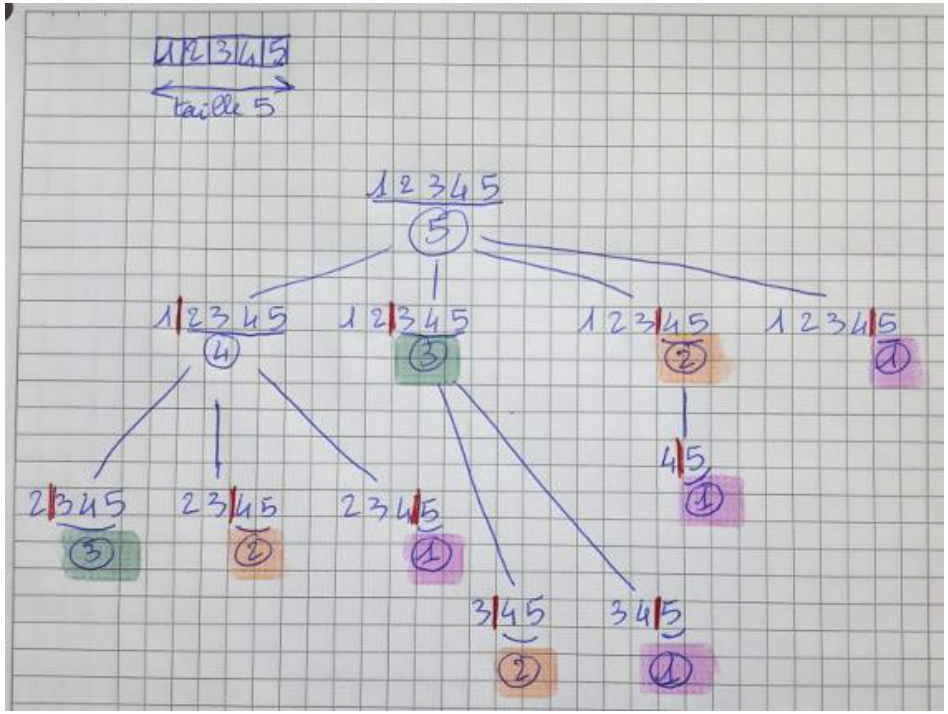
Calcul du meilleur gain



- À chaque étage, meilleur gain =

$$max \begin{cases} \text{Pour chaque découpe, (taille barre après découpe} \in [1; n]) \\ \quad \text{prix de la partie coupée} + \\ \quad \text{meilleur gain de ce que l'on peut tirer du reste} \\ \quad \text{de la découpe.} \end{cases}$$

Quelle efficacité ?



- Problème de taille 3 calculé 2 fois.
- Problème de taille 2 calculé 3 fois.
- Problème de taille 1 calculé 4 fois.
- Complexité exponentielle.
- ⇒ Éviter de recalculer.

Éviter de recalculer : version haut-bas

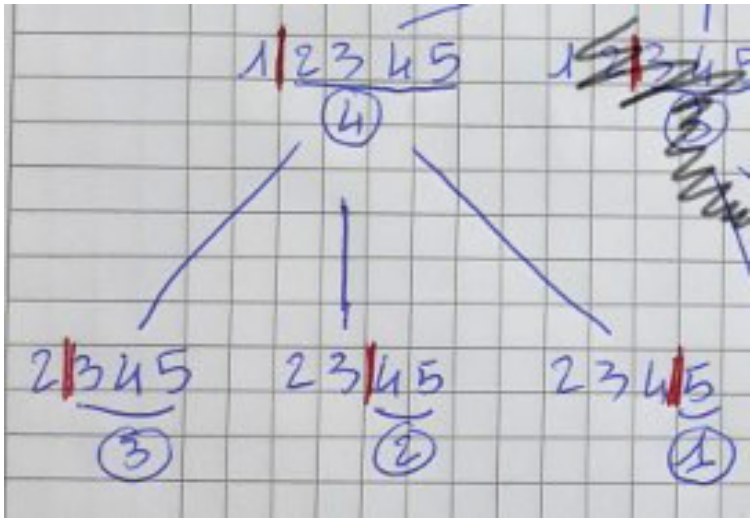
- Idée : mémoriser les calculs déjà faits pour les occurrences future
- Tableau de taille n.
- Avant de recalculer, regarder si résultat déjà dans le tableau.
- Si non, calculer et mémoriser dans le tableau. ⇒ Complexité : $O(n^2)$.

```
#define PSIZE (9)
int prices[PSIZE] = { 0, 2, 3, 8, 10, 13, 15, 16, 21 } ;
# Initially, no intermediate prices are recorded.
int memo_prices[PSIZE] = {-1, -1, -1, -1, -1, -1, -1, -1, -1} ;

int cut (int size) {
  if (size <= 0) return 0 ;
  if (memo_prices[size] >= 0) return memo_prices[size] ;
  int best = 0 ;          /* Best price, recursively. */
  int cut_size = 1 ;      /* Size of the removed part. */
  while (cut_size <= size) {  /* <= size alllows no cut. */
    best = max (best, prices[cut_size] + cut (size - cut_size));
    cut_size++ ;
  }
  memo_prices[size] = best ;
  return best ;
}
```

Éviter de recalculer : version bas-haut

- Version précédente : encore une récursion.
- Consommation mémoire au travers de la « pile d'exécution ».



$$\texttt{memo\_prices}[4] = max \begin{cases} \texttt{prices}[1] + \texttt{memo\_prices}[3] \\ \texttt{prices}[2] + \texttt{memo\_prices}[2] \\ \texttt{prices}[3] + \texttt{memo\_prices}[1] \\ \texttt{prices}[4] + \texttt{memo\_prices}[0] \end{cases}$$

- $\texttt{memo\_prices}[i] = max_{j=1 \text{ à } i} \left\{ \texttt{prices}[j] + \texttt{memo\_prices}[i-j] \right\}$
- Idée : remplir `memo_prices` pour i de 0 à n.
- $\Rightarrow$ 2 boucles ($O(n^2)$) mais récursion disparue.

Éviter de recalculer : version bas-haut

- $\texttt{memo\_prices}[i] = max_{j=1 \text{ à } i} \left\{ \texttt{prices}[j] + \texttt{memo\_prices}[i-j] \right\}$

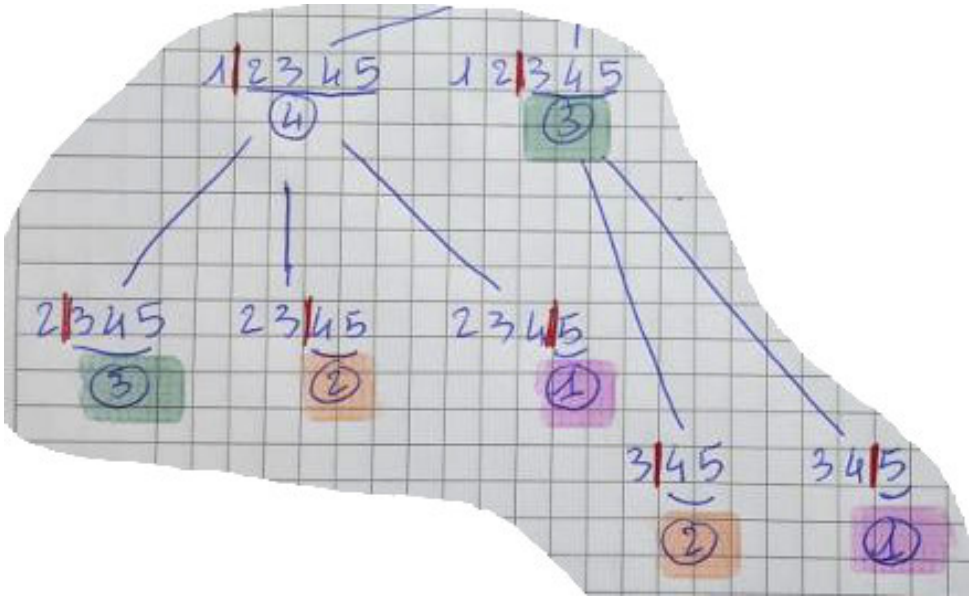
```
int prices[PSIZE] = { 0, 2, 3, 8, 10, 13, 15, 16, 21 } ;
int memo_prices[PSIZE] = { -1, -1, -1, -1, -1, -1, -1, -1, -1 };

int cut (int size) {
  for (int i = 0 ; i <= size; i++) {
    int best = 0 ;
    for (int j = 1; j <= i; j++)
      best = max (best, prices[j] + memo_prices[i - j]) ;
    memo_prices[i] = best ;
  }
  return memo_prices[size] ;
}
```

Programmation dynamique

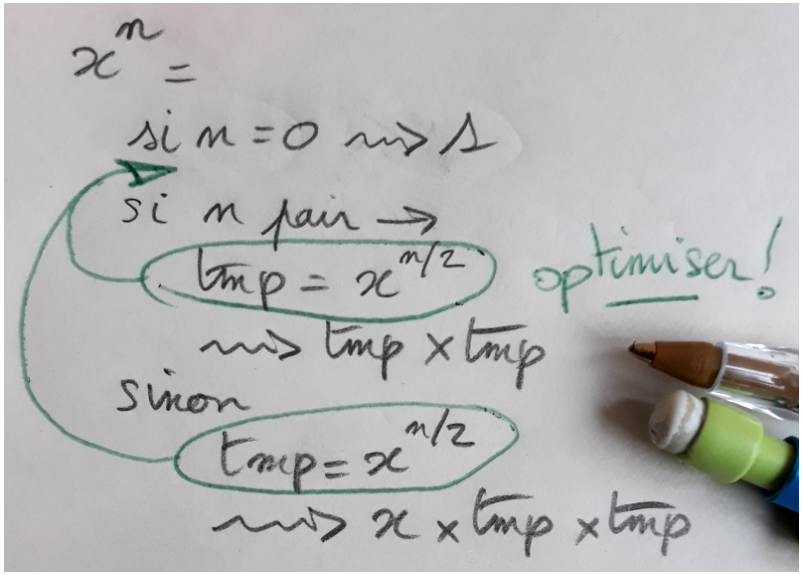
- Décomposer un problème en problèmes plus petits de même form
- Mémoriser les calculs intermédiaires.

- Exploration de toutes les configurations possibles.
- Fonctionne même si sous-problèmes pas indépendants.
  - Calculs pour taille = 4 et pour taille 3 non indépendants.



  - Chacun nécessite le calcul pour taille = 2.

Remarque sur l'algorithme x^n



- tmp = power (x, n / 2) puis retourner tmp ∗ tmp.
- On a directement évité de recalculer plusieurs fois la puissance.
- Le sous-problème pouvait être vu comme 2 dépendants (égaux).

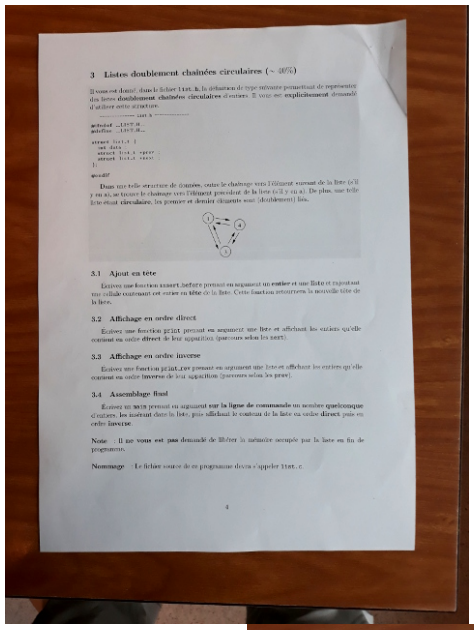
Paradigmes de programmation : algorithmes gloutons

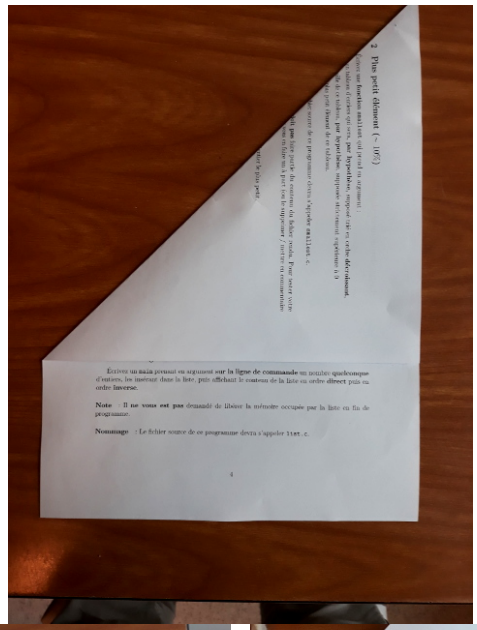
Le problème à résoudre

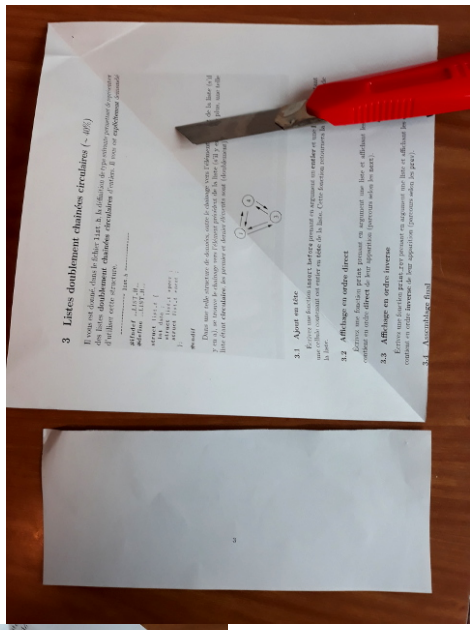
- Soit une feuille de dimensions $l_1 \times l_2$ (entières).
- Découper la feuille en carrés.
- Minimiser le nombre de carrés à découper.

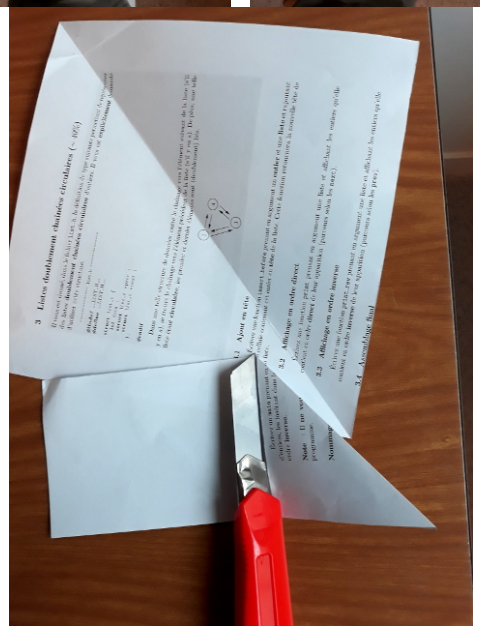
- Explorer toutes les découpes possibles ? Non !
- Essayer à chaque fois de faire le plus grand carré.
- À chaque étape, c'est la solution optimale.

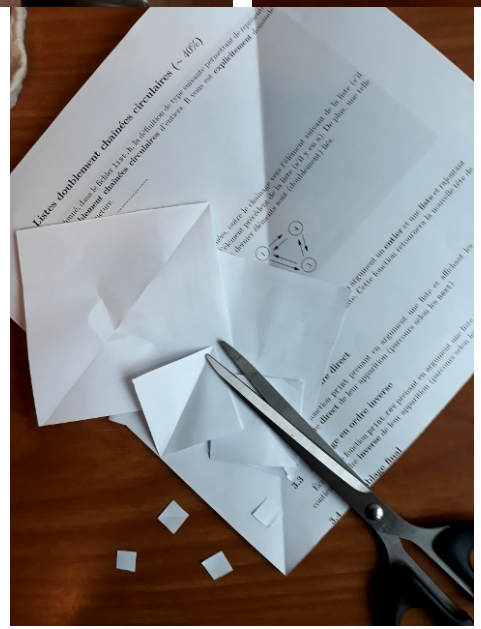
L'algorithme . . . sur papier



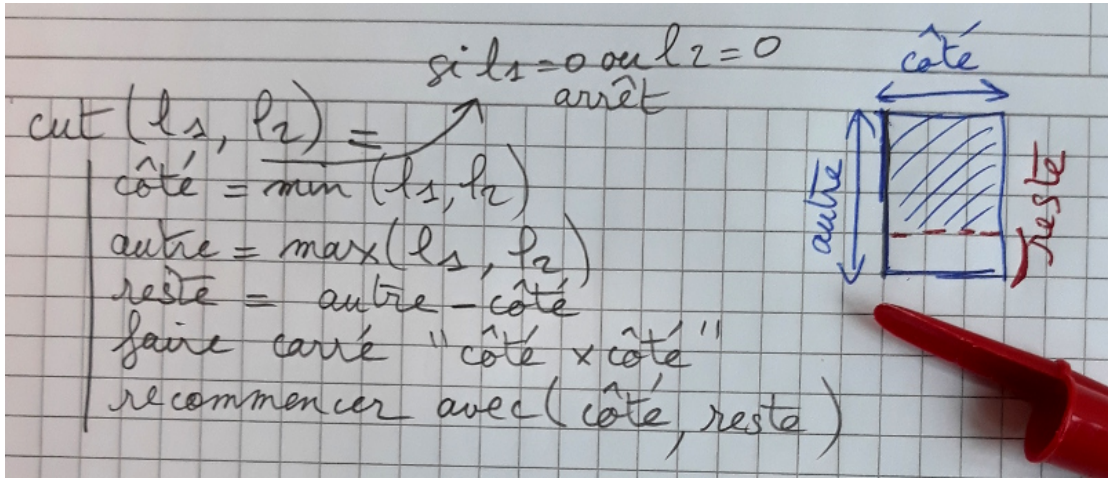








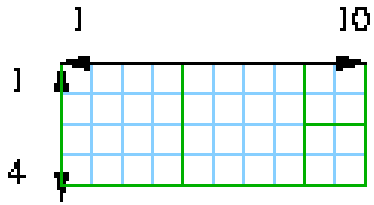
L'algorithme sans papier




```
void cut (int dim1, int dim2) {
  if ((dim1 != 0) && (dim2 != 0)) {
    int sq_size, rem ;
    if (dim1 < dim2) {
      sq_size = dim1 ; rem = dim2 - dim1 ;
    }
    else {
      sq_size = dim2 ; rem = dim1 - dim2 ;
    }
    printf ("Make %d x %d.\n", sq_size, sq_size) ;
    cut (sq_size, rem) ;
  }
}
```

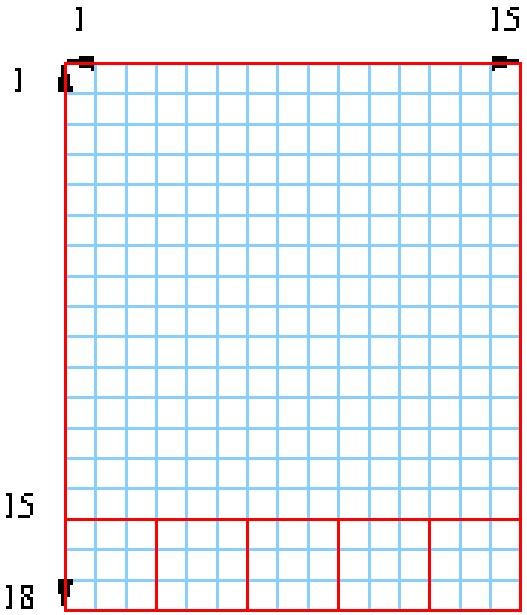
Optimalité ?

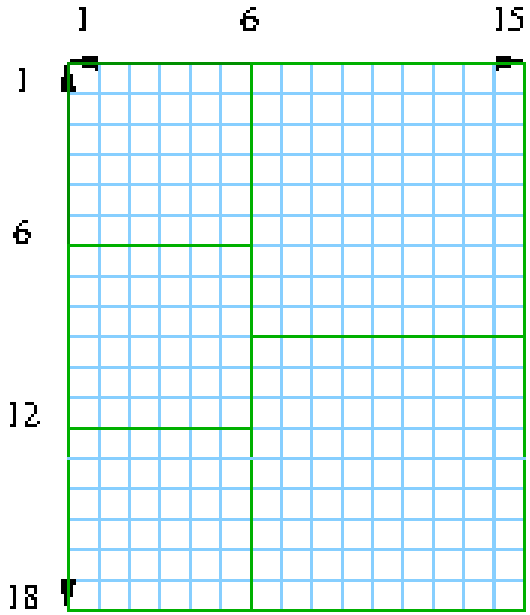
```
./cutsquare 10 4
Make  4 x 4
Make  4 x 4
Make  2 x 2
Make  2 x 2
```



- 10×4 : solution optimale trouvée par l'algorithme.

```
./cutsquare 15 18
Make  15 x 15
Make  3 x 3
Make  3 x 3
Make  3 x 3
Make  3 x 3
Make  3 x 3
```





- 15×18 : solution meilleure non trouvée par l'algorithme.

Algorithmes gloutons

- Problème posé, exploration de toutes les solutions : impossible.
- ⇒ Stratégie pour trouver une « bonne » solution au problème.
- À chaque étape, choix d'une solution optimale au problème local.
- Espoir : combinaison des optima locaux $\rightsquigarrow$ « bonne » solution au problème global.
- Rarement obtention d'une solution optimale globale.
- Algorithme glouton = heuristique.

Retour d'expérience en travaux pratiques

Suggestions pour l'organisation des TPs

- Expliciter les questions de réflexion préalables.
- Repousser l'implantation en fin de séance :
  - Évite la difficulté de synchronisation entre les exercices.
  - Évite de perdre les élèves (qui ne lâchent plus le clavier).
  - Évite d'ignorer les questions de réflexion suivantes.
- Co-construction des solutions : nécessité d'interactivité.

Exemple de trame

« Bla bla, le sujet, les informations »

Q1.1 Quelle structure de données choisir pour représenter un *truc* ?

Q1.2 Identifiez les grandes étapes du programme.

Q1.3 Quelles sont les traitements de la dernière étape ? Combien de fois sont-ils effectués ?

Q1.4 Comparez les opérations Q1.2-2 et Q1.3-2, que remarquez-vous ?

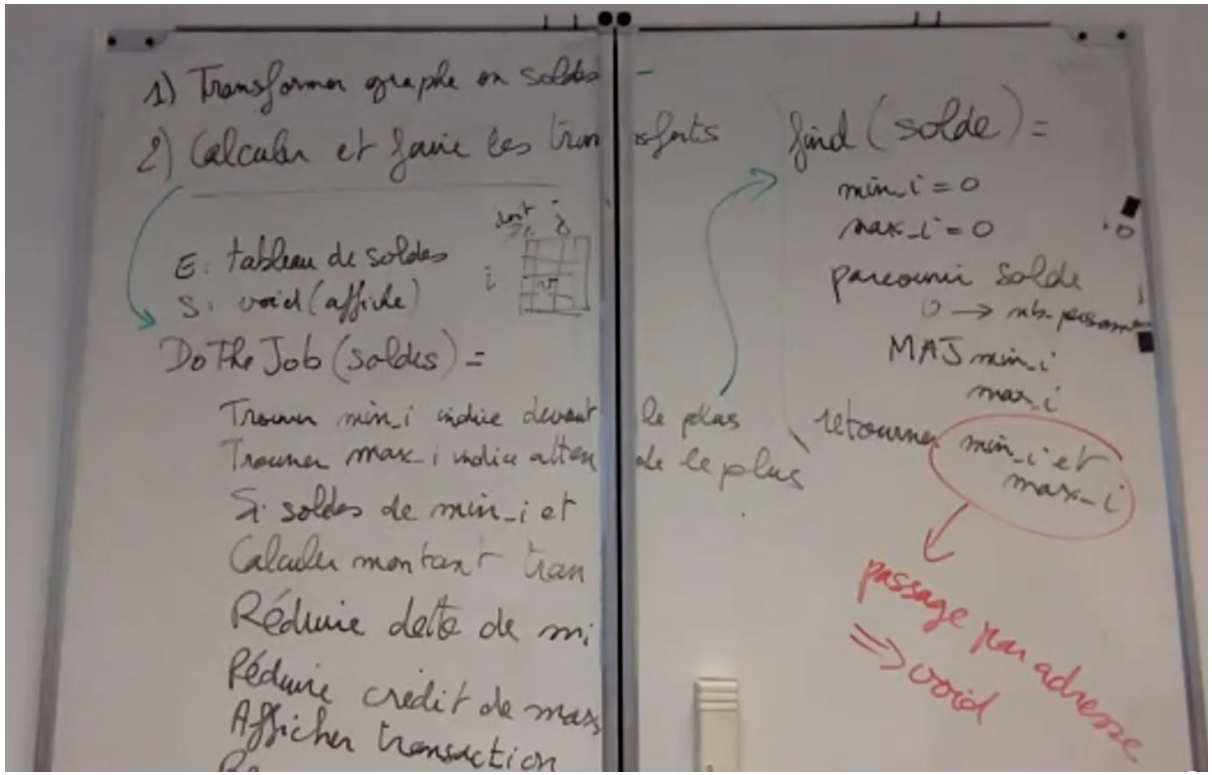
Q1.5 De l'algorithme esquissé en Q1.2, de quelles fonctions avez-vous besoin, que font-elles, quels sont leurs domaines ?

...

Q1.n : Implémentez en *langage bidule* toutes les fonctions identifiées et dont vous avez esquissé les algorithmes.

Mode d'interaction

- Utilisation importante du tableau.
- Dessins, schémas, essais.
- Raffinement successifs en pseudo-code si possible tous visibles et reliés.
- Code source final pas (forcément) nécessaire.



Avez-vous des questions ?