Problèmes du Millénaire: **P** vs **NP** ?

Xavier Provençal

3 juin 2022

Page web “P-versus-NP” par Gerhard J. Woeginger (lien).

La notion de *problème*

Définition

Un **problème** Q est une question qui contient un **paramètre** prenant valeur dans un ensemble infini dénombrable.

Lorsqu'on fixe la valeur du paramètre, on obtient une **instance** I et $Q(I) \in \{\texttt{oui}, \texttt{non}\}$ est la **réponse** à la question.

Exemples :

- L'entier n est-il premier ? $Q(5) = \texttt{oui}$, $Q(42) = \texttt{non}$.
- Le mot m apparait-il dans le texte t ?
- La conjecture de Riemann est-elle vraie ?
- L'entier n est-il un multiple de 1 ?

La notion de *problème*

Définition

Un **problème** Q est une question qui contient un **paramètre** prenant valeur dans un ensemble infini dénombrable.

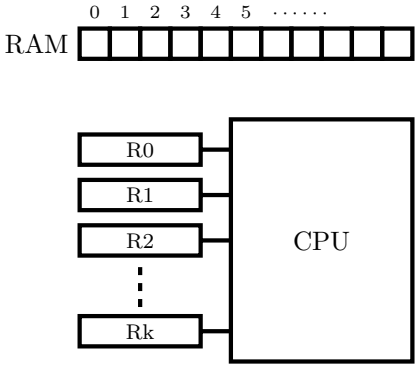
Lorsqu'on fixe la valeur du paramètre, on obtient une **instance** I et $Q(I) \in \{\texttt{oui}, \texttt{non}\}$ est la **réponse** à la question.

Exemples :

- L'entier n est-il premier ? $Q(5) = \texttt{oui}$, $Q(42) = \texttt{non}$.
- Le mot m apparait-il dans le texte t ?
- ~~La conjecture de Riemann est-elle vraie ?~~
- L'entier n est-il un multiple de 1 ?

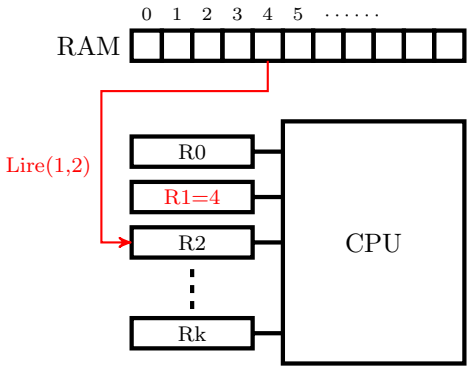
Le modèle RAM

- Chaque case mémoire et chaque registre contient une valeur.
- Nombre de valeurs possibles : K.
- Opérations possibles :



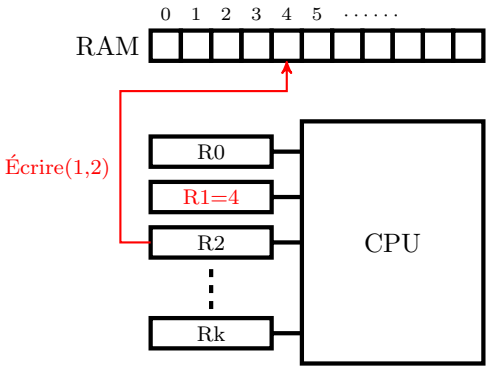
Le modèle RAM

- Chaque case mémoire et chaque registre contient une valeur.
- Nombre de valeurs possibles : K.
- Opérations possibles :
  - `Lire`(u, v) : la valeur dans la case mémoire identifiée par le registre Ru est copié dans le registre Rv.



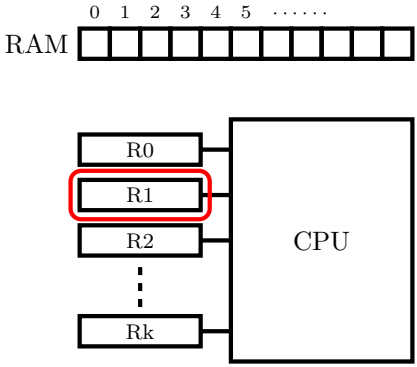
Le modèle RAM

- Chaque case mémoire et chaque registre contient une valeur.
- Nombre de valeurs possibles : K.
- Opérations possibles :
  - `Lire`(u, v) : la valeur dans la case mémoire identifiée par le registre Ru est copié dans le registre Rv.
  - `Écrire`(u, v) : la case mémoire identifiée par le registre Ru prend la valeur du registre Rv.



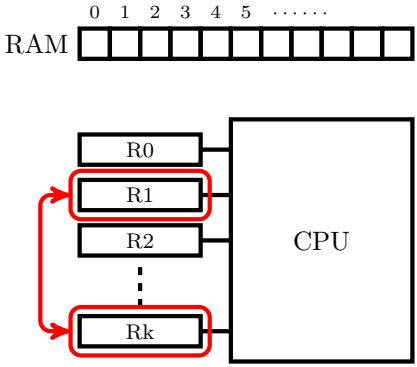
Le modèle RAM

- Chaque case mémoire et chaque registre contient une valeur.
- Nombre de valeurs possibles : K.
- Opérations possibles :
  - `Lire`(u, v) : la valeur dans la case mémoire identifiée par le registre Ru est copié dans le registre Rv.
  - `Écrire`(u, v) : la case mémoire identifiée par le registre Ru prend la valeur du registre Rv.
  - Modifier le contenu du registre Ri.



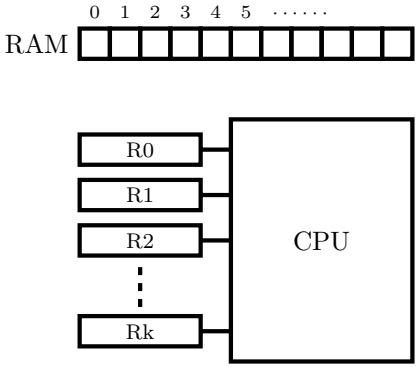
Le modèle RAM

- Chaque case mémoire et chaque registre contient une valeur.
- Nombre de valeurs possibles : K.
- Opérations possibles :
  - `Lire`(u, v) : la valeur dans la case mémoire identifiée par le registre Ru est copié dans le registre Rv.
  - `Écrire`(u, v) : la case mémoire identifiée par le registre Ru prend la valeur du registre Rv.
  - Modifier le contenu du registre Ri.
  - Modifier le contenu du registre Ri en fonction du registre Rj.



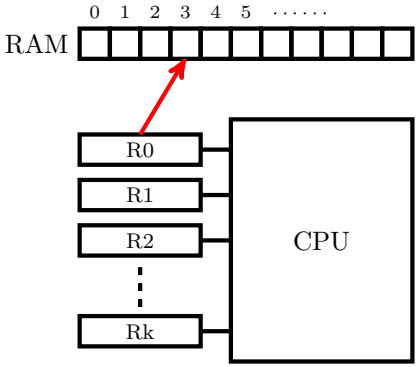
Le modèle RAM

- Chaque case mémoire et chaque registre contient une valeur.
- Nombre de valeurs possibles : K.
- Opérations possibles :
  - `Lire`(u, v) : la valeur dans la case mémoire identifiée par le registre Ru est copié dans le registre Rv.
  - `Écrire`(u, v) : la case mémoire identifiée par le registre Ru prend la valeur du registre Rv.
  - Modifier le contenu du registre Ri.
  - Modifier le contenu du registre Ri en fonction du registre Rj.
  - Terminer l'exécution.



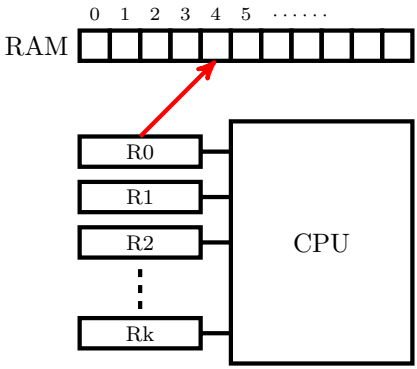
Le modèle RAM

- Chaque case mémoire et chaque registre contient une valeur.
- Nombre de valeurs possibles : K.
- Opérations possibles :
  - `Lire`(u, v) : la valeur dans la case mémoire identifiée par le registre Ru est copié dans le registre Rv.
  - `Écrire`(u, v) : la case mémoire identifiée par le registre Ru prend la valeur du registre Rv.
  - Modifier le contenu du registre Ri.
  - Modifier le contenu du registre Ri en fonction du registre Rj.
  - Terminer l'exécution.
- Registre R0 : prochaine instruction.



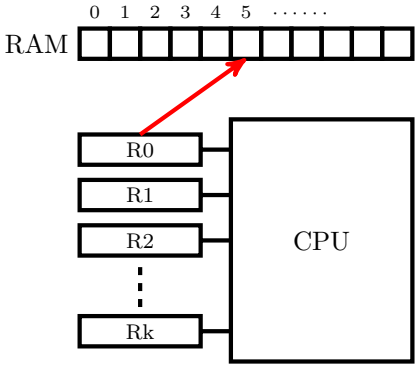
Le modèle RAM

- Chaque case mémoire et chaque registre contient une valeur.
- Nombre de valeurs possibles : K.
- Opérations possibles :
  - `Lire`(u, v) : la valeur dans la case mémoire identifiée par le registre Ru est copié dans le registre Rv.
  - `Écrire`(u, v) : la case mémoire identifiée par le registre Ru prend la valeur du registre Rv.
  - Modifier le contenu du registre Ri.
  - Modifier le contenu du registre Ri en fonction du registre Rj.
  - Terminer l'exécution.
- Registre R0 : prochaine instruction.



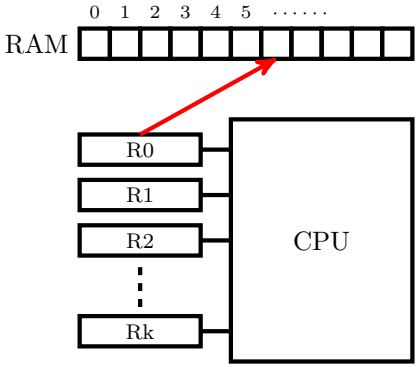
Le modèle RAM

- Chaque case mémoire et chaque registre contient une valeur.
- Nombre de valeurs possibles : K.
- Opérations possibles :
  - `Lire`(u, v) : la valeur dans la case mémoire identifiée par le registre Ru est copié dans le registre Rv.
  - `Écrire`(u, v) : la case mémoire identifiée par le registre Ru prend la valeur du registre Rv.
  - Modifier le contenu du registre Ri.
  - Modifier le contenu du registre Ri en fonction du registre Rj.
  - Terminer l'exécution.
- Registre R0 : prochaine instruction.



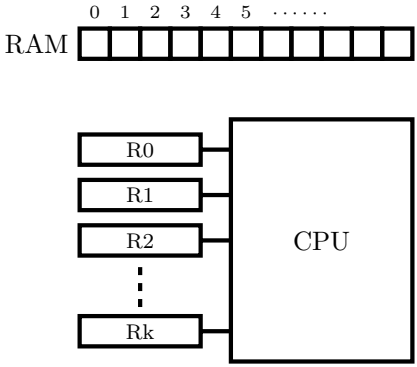
Le modèle RAM

- Chaque case mémoire et chaque registre contient une valeur.
- Nombre de valeurs possibles : K.
- Opérations possibles :
  - `Lire`(u, v) : la valeur dans la case mémoire identifiée par le registre Ru est copié dans le registre Rv.
  - `Écrire`(u, v) : la case mémoire identifiée par le registre Ru prend la valeur du registre Rv.
  - Modifier le contenu du registre Ri.
  - Modifier le contenu du registre Ri en fonction du registre Rj.
  - Terminer l'exécution.
- Registre R0 : prochaine instruction.



Le modèle RAM

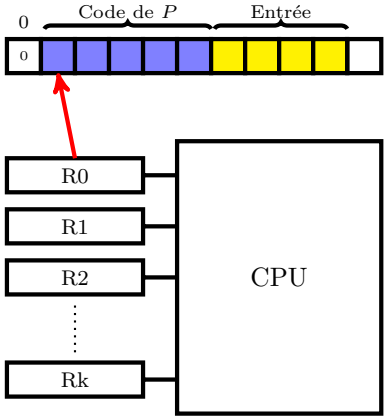
- Chaque case mémoire et chaque registre contient une valeur.
- Nombre de valeurs possibles : K.
- Opérations possibles :
  - `Lire`(u, v) : la valeur dans la case mémoire identifiée par le registre Ru est copié dans le registre Rv.
  - `Écrire`(u, v) : la case mémoire identifiée par le registre Ru prend la valeur du registre Rv.
  - Modifier le contenu du registre Ri.
  - Modifier le contenu du registre Ri en fonction du registre Rj.
  - Terminer l'exécution.
- Registre R0 : prochaine instruction.



- **Chaque opération est exécutée en un temps borné par une constante.**

Initialisation :

- La case mémoire 0 est initialisée à 0.
- Le code du programme P est inscrit en mémoire à partir de la case 1.
- Des données, appelées **entrée**, sont inscrites en mémoire à la suite.
- Le registre $R0$ est initialisé à 1.



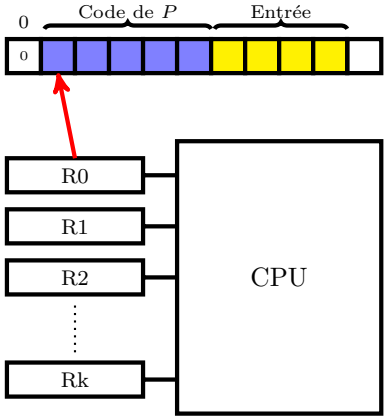
Le modèle RAM

Initialisation :

- La case mémoire 0 est initialisée à 0.
- Le code du programme P est inscrit en mémoire à partir de la case 1.
- Des données, appelées **entrée**, sont inscrites en mémoire à la suite.
- Le registre $R0$ est initialisé à 1.

Trois cas possibles :

- Termine et la case 0 vaut 1 : **acceptation**.
- Termine et la case 0 ne vaut pas 1 : **refus**.
- Ne termine pas.



Définition

Le programme P **résout** le problème Q si

- le programme P termine peu importe l'entrée,
- pour toute instance I de Q, $P(I) = Q(I)$.

Un **algorithme** est une *description* d'un programme qui résout un certain problème Q.

Exemple

Définition

Une liste L de taille n est triée, si

$$\forall i, j \in \{0, 1, \ldots, n-1\}, \quad i < j \rightarrow L[i] \leq L[j].$$

Problème

La liste L est-elle triée ?

Algorithme

fonction `estTriée`(L : liste de taille n)
- **pour** i *de* 0 *à* $n-1$ **faire**
  - **pour** j *de* 0 *à* $n-1$ **faire**
    - **si** $i < j$ **et** $L[i] > L[j]$ **alors**
      - **retourner** `non`
- **retourner** `oui`

Complexité d'un algorithme

Définition

La **fonction de complexité** $f(n)$ d'un algorithme A est le nombre d'opérations effectuées par l'algorithme A pour une entrée de **taille** n, au pire cas.

Exemple :

```
fonction estTriée(L : liste de taille n)
    pour i de 0 à n − 1 faire
        pour j de 0 à n − 1 faire
            si i < j et L[i] > L[j] alors
                retourner non
    retourner oui
```

Complexité d'un algorithme

Définition

La **fonction de complexité** $f(n)$ d'un algorithme A est le nombre d'opérations effectuées par l'algorithme A pour une entrée de **taille** n, au pire cas.

Exemple :

fonction `estTriée`(L : liste de taille n)
- **pour** i *de* 0 *à* $n-1$ **faire**
  - **pour** j *de* 0 *à* $n-1$ **faire**
    - **si** $i < j$ **et** $L[i] > L[j]$ **alors** c_1
      - **retourner** `non`
- **retourner** `oui`

Complexité d'un algorithme

Définition

La **fonction de complexité** $f(n)$ d'un algorithme A est le nombre d'opérations effectuées par l'algorithme A pour une entrée de **taille** n, au pire cas.

Exemple :

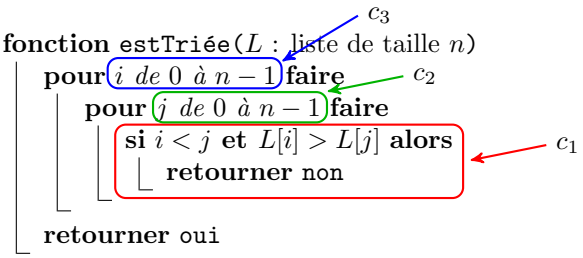
fonction `estTriée`(L : liste de taille n)
- **pour** i *de* 0 *à* $n-1$ **faire** — c_2
  - **pour** j *de* 0 *à* $n-1$ **faire**
    - **si** $i < j$ **et** $L[i] > L[j]$ **alors** — c_1
      - **retourner** `non`
- **retourner** `oui`

Complexité d'un algorithme

Définition

La **fonction de complexité** $f(n)$ d'un algorithme A est le nombre d'opérations effectuées par l'algorithme A pour une entrée de **taille** n, au pire cas.

Exemple :

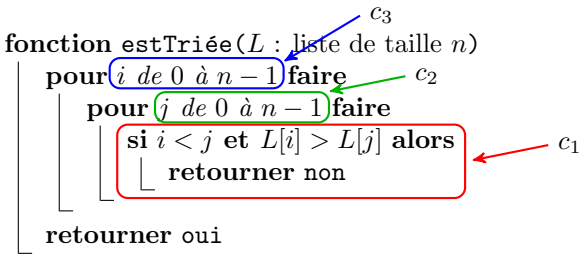


Complexité d'un algorithme

Définition

La **fonction de complexité** $f(n)$ d'un algorithme A est le nombre d'opérations effectuées par l'algorithme A pour une entrée de **taille** n, au pire cas.

Exemple :



$$f(n) \approx n(c_3 + n(c_2 + c_1)).$$

Notation grand-O

Définition

Un fonction de complexité $f(n)$ est dans $O(g(n))$ si

$$\exists k \geq 0, \exists C > 0, \quad n > k \rightarrow f(n) \leq Cg(n).$$

De manière équivalente : $f(n) \in O(g(n)) \leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$.

Notation grand-O

Définition

Un fonction de complexité $f(n)$ est dans $O(g(n))$ si

$$\exists k \geq 0, \exists C > 0, \quad n > k \rightarrow f(n) \leq Cg(n).$$

De manière équivalente : $f(n) \in O(g(n)) \leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.

Grand-O crée une hiérarchie de classes de complexité :

$$O(1) \subseteq O(\log(n)) \subseteq O(n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n) \subseteq O(3^n) \subseteq O(n!) \subseteq \cdots$$

Notation grand-O

Définition

Un fonction de complexité $f(n)$ est dans $O(g(n))$ si

$$\exists k \geq 0, \exists C > 0, \quad n > k \rightarrow f(n) \leq Cg(n).$$

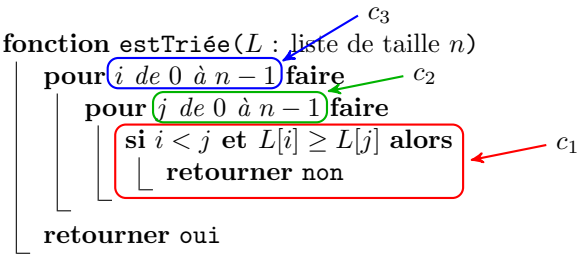
De manière équivalente : $f(n) \in O(g(n)) \leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.

Grand-O crée une hiérarchie de classes de complexité :

$$O(1) \subseteq O(\log(n)) \subseteq O(n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n) \subseteq O(3^n) \subseteq O(n!) \subseteq \cdots$$

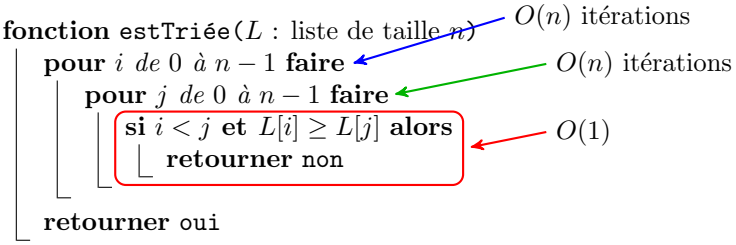
Classe de complexité	Comportement lorsque n est *grand*
$O(n)$	$f(2n) \leq 2f(n)$
$O(n^2)$	$f(2n) \leq 4f(n)$
$O(n^d)$	$f(2n) \leq 2^d f(n)$
$O(2^n)$	$f(n+1) \leq 2f(n)$

Exemple :



$$f(n) \approx n(c_3 + n(c_2 + c_1)).$$

Exemple :



$$f(n) \in O(n^2).$$

Temps polynomial

Définition

Un algorithme A avec fonction de complexité $f(n)$ est **polynomial**, s'il existe une constante $d > 0$ telle que $f(n) \in O(n^d)$.

Polynomial

pour i_1 *de* 1 *à* n **faire**
  pour i_2 *de* 1 *à* n **faire**
    $\ddots$
    pour i_k *de* 1 *à* n **faire**
      $\vdots$
pour j_1 *de* 1 *à* n^{d_1} **faire**
  pour j_2 *de* 1 *à* n^{d_2} **faire**
    $\ddots$
    pour j_l *de* 1 *à* n^{d_l} **faire**
      $\vdots$
  $\vdots$

Non polynomial

pour chaque entier de 1 à 10^n **faire**
  $\vdots$

pour chaque permutation de n objets **faire**
  $\vdots$

pour chaque suite de bits de longueur n **faire**
  $\vdots$

Complexité d'un problème

Définition

La **complexité** d'un problème Q est la complexité du *meilleur* algorithme qui le résout.

Exemple : "La liste L est-elle triée ?"

Solution naïve

```
fonction estTriée(L : liste de taille n)
    pour i de 0 à n − 1 faire
        pour j de 0 à n − 1 faire
            si i < j et L[i] > L[j] alors
                retourner non
    retourner oui
```

Le problème est dans $O(n^2)$.

Par la transitivité de $\leq$

```
fonction estTriée(L : liste de taille n)
    pour i de 0 à n − 2 faire
        si L[i] > L[i + 1] alors
            retourner non
    retourner oui
```

Le problème est dans $O(n)$.

La classe **P**

Définition

La classe **P** est l'ensemble des problèmes de complexité polynomiale.

- Pour prouver $Q \in \mathbf{P}$, il *suffit* de fournir un algorithme polynomial pour le résoudre.
- Pour prouver $Q \notin \mathbf{P}$, il faut prouver qu'il n'en existe pas.

Exemples :

- Le problème "*est triée*" $\in \mathbf{P}$.
- Le problème de l'arrêt $\notin \mathbf{P}$.

Exemple

Problème

L'entier x écrit sur n chiffres est-il premier ?

Solution naïve

```
fonction estPremier(x : entier de taille n)
    pour i de 2 à x − 1 faire
        si x mod i == 0 alors
            retourner non
    retourner oui
```

Le problème est dans $O(n^2 10^n)$.

Exemple

Problème

L'entier x écrit sur n chiffres est-il premier ?

Solution naïve

```
fonction estPremier(x : entier de taille n)
    pour i de 2 à ⌊√x⌋ faire
        si x mod i == 0 alors
            retourner non
    retourner oui
```

Le problème est dans $O(n^2 10^{n/2}) \approx O(n^2 3^n)$.

Exemple

Problème

L'entier x écrit sur n chiffres est-il premier ?

Solution naïve

```
fonction estPremier(x : entier de taille n)
    pour i de 2 à ⌊√x⌋ faire
        si x mod i == 0 alors
            retourner non
    retourner oui
```

Le problème est dans $O(n^2 10^{n/2}) \approx O(n^2 3^n)$.

Test AKS

- Publié en 2002 par **A**grawal, **K**ayal et **S**axena.
- Test de primalité en $O(n^{12})$.

Le problème est dans $O(n^{12})$.

Exemple

Problème

L'entier x écrit sur n chiffres est-il premier ?

Solution naïve

fonction `estPremier`(x : entier de taille n)
  pour i *de* 2 *à* $\lfloor\sqrt{x}\rfloor$ **faire**
    si $x \mod i == 0$ **alors**
      retourner `non`
  retourner `oui`

Le problème est dans $O(n^2 10^{n/2}) \approx O(n^2 3^n)$.

Test AKS

- Publié en 2002 par **A**grawal, **K**ayal et **S**axena.
- Test de primalité en $O(n^{12})$.

Le problème est dans $O(n^{12})$.

Le problème "*est premier*" est donc dans **P**.

Définition

La classe **NP** est l'ensemble de problèmes pour lesquels il existe un *algorithme de vérification* en temps polynomial.

Définition

La classe **NP** est l'ensemble de problèmes pour lesquels il existe un *algorithme de vérification* en temps polynomial.

Définition

Un **algorithme de vérification** pour le problème Q est un algorithme A tel que, pour toute instance I de Q :

- Si $Q(I) = \texttt{oui}$, alors il existe c un **certificat** pour I.
- $A(I, c) = \begin{cases} \texttt{oui} & \text{si } Q(I) = \texttt{oui} \text{ et } c \text{ est un } \textbf{certificat} \text{ valide,} \\ \texttt{oui ou non} & \text{si } Q(I) = \texttt{oui} \text{ et } c \text{ est un } \textbf{certificat} \text{ invalide,} \\ \texttt{non} & \text{si } Q(I) = \texttt{non.} \end{cases}$

Définition

La classe **NP** est l'ensemble de problèmes pour lesquels il existe un *algorithme de vérification* en temps polynomial.

Définition

Un **algorithme de vérification** pour le problème Q est un algorithme A tel que, pour toute instance I de Q :

- Si $Q(I) = \texttt{oui}$, alors il existe c un **certificat** pour I.
- $A(I, c) = \begin{cases} \texttt{oui} & \text{si } Q(I) = \texttt{oui} \text{ et } c \text{ est un } \textbf{certificat} \text{ valide,} \\ \texttt{oui ou non} & \text{si } Q(I) = \texttt{oui} \text{ et } c \text{ est un } \textbf{certificat} \text{ invalide,} \\ \texttt{non} & \text{si } Q(I) = \texttt{non.} \end{cases}$

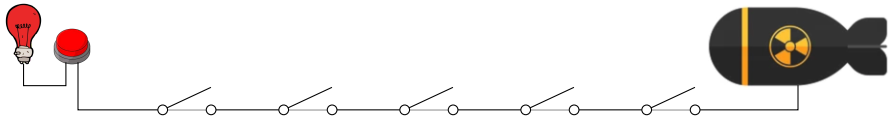
Définition

La classe **NP** est l'ensemble de problèmes pour lesquels il existe un *algorithme de vérification* en temps polynomial.

Définition

Un **algorithme de vérification** pour le problème Q est un algorithme A tel que, pour toute instance I de Q :

- Si $Q(I) = \texttt{oui}$, alors il existe c un **certificat** pour I.
- $A(I, c) = \begin{cases} \texttt{oui} & \text{si } Q(I) = \texttt{oui} \text{ et } c \text{ est un } \textbf{certificat} \text{ valide,} \\ \texttt{oui ou non} & \text{si } Q(I) = \texttt{oui} \text{ et } c \text{ est un } \textbf{certificat} \text{ invalide,} \\ \texttt{non} & \text{si } Q(I) = \texttt{non.} \end{cases}$



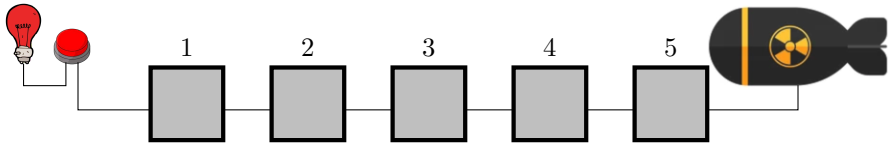
Définition

La classe **NP** est l'ensemble de problèmes pour lesquels il existe un *algorithme de vérification* en temps polynomial.

Définition

Un **algorithme de vérification** pour le problème Q est un algorithme A tel que, pour toute instance I de Q :

- Si $Q(I) = \texttt{oui}$, alors il existe c un **certificat** pour I.
- $A(I, c) = \begin{cases} \texttt{oui} & \text{si } Q(I) = \texttt{oui} \text{ et } c \text{ est un } \textbf{certificat} \text{ valide,} \\ \texttt{oui ou non} & \text{si } Q(I) = \texttt{oui} \text{ et } c \text{ est un } \textbf{certificat} \text{ invalide,} \\ \texttt{non} & \text{si } Q(I) = \texttt{non.} \end{cases}$



Question : est-ce qu'au moins un interrupteur est ouvert ?
Certificat : le numéro de la boîte.

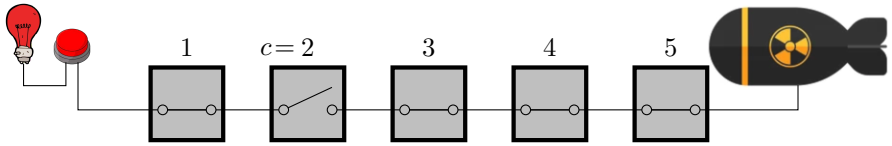
Définition

La classe **NP** est l'ensemble de problèmes pour lesquels il existe un *algorithme de vérification* en temps polynomial.

Définition

Un **algorithme de vérification** pour le problème Q est un algorithme A tel que, pour toute instance I de Q :

- Si $Q(I) = \texttt{oui}$, alors il existe c un **certificat** pour I.
- $A(I, c) = \begin{cases} \texttt{oui} & \text{si } Q(I) = \texttt{oui} \text{ et } c \text{ est un } \textbf{certificat} \text{ valide,} \\ \texttt{oui ou non} & \text{si } Q(I) = \texttt{oui} \text{ et } c \text{ est un } \textbf{certificat} \text{ invalide,} \\ \texttt{non} & \text{si } Q(I) = \texttt{non.} \end{cases}$



Question : est-ce qu'au moins un interrupteur est ouvert ?
Certificat : le numéro de la boîte.

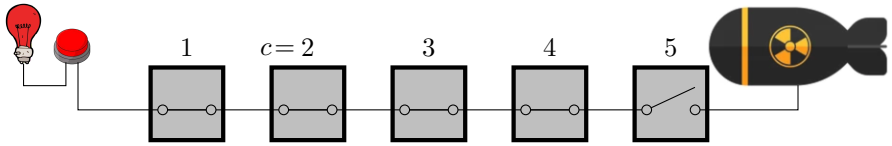
Définition

La classe **NP** est l'ensemble de problèmes pour lesquels il existe un *algorithme de vérification* en temps polynomial.

Définition

Un **algorithme de vérification** pour le problème Q est un algorithme A tel que, pour toute instance I de Q :

- Si $Q(I) = \texttt{oui}$, alors il existe c un **certificat** pour I.
- $A(I, c) = \begin{cases} \texttt{oui} & \text{si } Q(I) = \texttt{oui} \text{ et } c \text{ est un } \textbf{certificat} \text{ valide,} \\ \texttt{oui ou non} & \text{si } Q(I) = \texttt{oui} \text{ et } c \text{ est un } \textbf{certificat} \text{ invalide,} \\ \texttt{non} & \text{si } Q(I) = \texttt{non.} \end{cases}$



Question : est-ce qu'au moins un interrupteur est ouvert ?
Certificat : le numéro de la boîte.

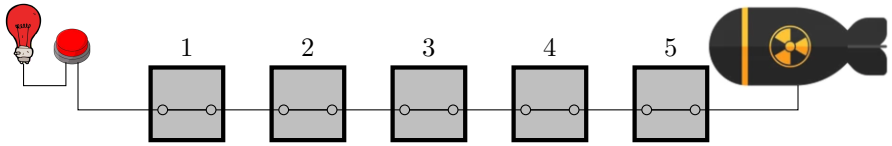
Définition

La classe **NP** est l'ensemble de problèmes pour lesquels il existe un *algorithme de vérification* en temps polynomial.

Définition

Un **algorithme de vérification** pour le problème Q est un algorithme A tel que, pour toute instance I de Q :

- Si $Q(I) = \texttt{oui}$, alors il existe c un **certificat** pour I.
- $A(I, c) = \begin{cases} \texttt{oui} & \text{si } Q(I) = \texttt{oui} \text{ et } c \text{ est un } \textbf{certificat} \text{ valide,} \\ \texttt{oui ou non} & \text{si } Q(I) = \texttt{oui} \text{ et } c \text{ est un } \textbf{certificat} \text{ invalide,} \\ \texttt{non} & \text{si } Q(I) = \texttt{non.} \end{cases}$



Question : est-ce qu'au moins un interrupteur est ouvert ?
Certificat : le numéro de la boîte.

Exemple de problème dans **NP**

Problème

L'entier x écrit sur n chiffres est-il composé ?

- Le certificat c est un divisieur de x.

Exemple de problème dans **NP**

Problème

L'entier x écrit sur n chiffres est-il composé ?

- Le certificat c est un divisieur de x.
- Algorithme de vérification pour le problème "*est composé*" :

  fonction `testDiviseur`(x : entier de taille n, c : un entier)

  retourner $(c \geq 2)$ et $(c < x)$ et $(x \mod c == 0)$;

- Complexité : $O(n^2)$, le problème "*est composé*" est donc dans **NP**.

Exemple de problème dans **NP**

Problème

L'entier x écrit sur n chiffres est-il composé ?

- Le certificat c est un divisieur de x.
- Algorithme de vérification pour le problème "*est composé*" :

  fonction `testDiviseur`(x : entier de taille n, c : un entier)

  retourner $(c \geq 2)$ et $(c < x)$ et $(x \mod c == 0)$;

- Complexité : $O(n^2)$, le problème "*est composé*" est donc dans **NP**.

- On pourrait aussi utiliser le test AKS pour *vérifier* une réponse à "*est composé*", sans utiliser le certificat.

Exemple de problème dans **NP**

Problème

L'entier x écrit sur n chiffres est-il composé ?

- Le certificat c est un divisieur de x.
- Algorithme de vérification pour le problème "*est composé*" :

  fonction `testDiviseur`(x : entier de taille n, c : un entier)

  retourner $(c \geq 2)$ et $(c < x)$ et $(x \mod c == 0)$;

- Complexité : $O(n^2)$, le problème "*est composé*" est donc dans **NP**.

- On pourrait aussi utiliser le test AKS pour *vérifier* une réponse à "*est composé*", sans utiliser le certificat.
- En généralisant cette approche à tous les problèmes de **P**, on conclut $\mathbf{P} \subseteq \mathbf{NP}$.

La question **P** = **NP** ?

Question

Les ensembles **P** et **NP** sont-il égaux ?

Autrement dit, existe-t-il un problème Q tel que

- Il **existe** un algorithme de **vérification** en temps polynomial pour Q.
- Il **n'existe pas** d'algorithme en temps polynomial pour résoudre Q.

La question $\mathbf{P} = \mathbf{NP}$?

Question

Les ensembles **P** et **NP** sont-il égaux ?

Autrement dit, existe-t-il un problème Q tel que

- Il **existe** un algorithme de **vérification** en temps polynomial pour Q.
- Il **n'existe pas** d'algorithme en temps polynomial pour résoudre Q.

- Pour montrer que $\mathbf{P} = \mathbf{NP}$, il faut montrer que pour tout problème de **NP**, il existe un algorithme en temps polynomial.
- Pour montrer que $\mathbf{P} \neq \mathbf{NP}$, il faut montrer qu'il existe un problème de **NP** pour lequel il n'existe pas d'algorithme en temps polynomial.

Page Wikipedia de *Mirifici Logarithmorum Canonis Descriptio* (lien).

Réduction

Définition

Soit Q_1 et Q_2 deux problèmes. Un **algorithme de réduction** de Q_1 vers Q_2, est une fonction $r : \{0,1\}^* \longrightarrow \{0,1\}^*$ telle que pour toute instance I de Q_1,

$$Q_1(I) = Q_2(r(I)).$$

Réduction

Définition

Soit Q_1 et Q_2 deux problèmes. Un **algorithme de réduction** de Q_1 vers Q_2, est une fonction $r : \{0,1\}^* \longrightarrow \{0,1\}^*$ telle que pour toute instance I de Q_1,

$$Q_1(I) = Q_2(r(I)).$$

- Soit A_2 un algorithme qui résout Q_2 avec une complexité $f_2(n)$, on construit un algorithme A_1 pour résoudre Q_1 :

fonction A_1(*I : instance de Q_1*)
    retourner $A_2(r(I))$

Réduction

Définition

Soit Q_1 et Q_2 deux problèmes. Un **algorithme de réduction** de Q_1 vers Q_2, est une fonction $r : \{0,1\}^* \longrightarrow \{0,1\}^*$ telle que pour toute instance I de Q_1,

$$Q_1(I) = Q_2(r(I)).$$

- Soit A_2 un algorithme qui résout Q_2 avec une complexité $f_2(n)$, on construit un algorithme A_1 pour résoudre Q_1 :

  fonction A_1(*I : instance de Q_1*)

  retourner $A_2(r(I))$

- La complexité de A_1 est $f_1(n) \in O\left(f_2\left(f_r(n)\right)\right)$ où $f_r(n)$ est la fonction de complexité de r.

Réduction

Définition

Soit Q_1 et Q_2 deux problèmes. Un **algorithme de réduction** de Q_1 vers Q_2, est une fonction $r : \{0,1\}^* \longrightarrow \{0,1\}^*$ telle que pour toute instance I de Q_1,

$$Q_1(I) = Q_2(r(I)).$$

- Soit A_2 un algorithme qui résout Q_2 avec une complexité $f_2(n)$, on construit un algorithme A_1 pour résoudre Q_1 :

  fonction A_1(*I* : instance de Q_1)

  retourner $A_2(r(I))$

- La complexité de A_1 est $f_1(n) \in O\left(f_2\left(f_r(n)\right)\right)$ où $f_r(n)$ est la fonction de complexité de r.

- Si f_r et f_2 sont des polynômes, alors f_1 est un polynôme.

Hiérarchie via la réduction

soient Q_1, Q_2 deux problèmes tels qu'il existe r une réduction polynomiale de Q_1 vers Q_2.

- Si $Q_2 \in \mathbf{P}$ alors $Q_1 \in \mathbf{P}$.
- Si $Q_1 \notin \mathbf{P}$ alors $Q_2 \notin \mathbf{P}$.
- On note alors $Q_1 \preceq Q_2$.

Le problème SAT

Problème (SAT)

L'expression booléenne X est-elle satisfaisable ?

Une expression booléenne est formée de :

- variables booléennes : $x_1, x_2, \ldots, x_n \in \{\textbf{vrai}, \textbf{faux}\}$.
- opérateurs :
  - $\neg X$: négation, inverse la valeur de X.
  - $X \wedge Y$: et, **vrai** ssi X et Y sont **vrai**.
  - $X \vee Y$: ou, **faux** ssi X et Y sont **faux**.

Le problème SAT

Problème (SAT)

L'expression booléenne X est-elle satisfaisable ?

Une expression booléenne est formée de :

- variables booléennes : $x_1, x_2, \ldots, x_n \in \{\textbf{vrai}, \textbf{faux}\}$.
- opérateurs :
  - $\neg X$: négation, inverse la valeur de X.
  - $X \wedge Y$: et, **vrai** ssi X et Y sont **vrai**.
  - $X \vee Y$: ou, **faux** ssi X et Y sont **faux**.

Définition

L'expression booléenne X est **satisfaisable** s'il existe une affectation de ses variables telle que X est **vrai**.

Le problème SAT

Problème (SAT)

L'expression booléenne X est-elle satisfaisable ?

Une expression booléenne est formée de :

- variables booléennes : $x_1, x_2, \ldots, x_n \in \{\textbf{vrai}, \textbf{faux}\}$.
- opérateurs :
  - $\neg X$: négation, inverse la valeur de X.
  - $X \wedge Y$: et, **vrai** ssi X et Y sont **vrai**.
  - $X \vee Y$: ou, **faux** ssi X et Y sont **faux**.

Définition

L'expression booléenne X est **satisfaisable** s'il existe une affectation de ses variables telle que X est **vrai**.

- $x_1 \wedge \neg x_2$ est satisfaite avec $x_1 = \textbf{vrai}$ et $x_2 = \textbf{faux}$.

Le problème SAT

Problème (SAT)

L'expression booléenne X est-elle satisfaisable ?

Une expression booléenne est formée de :

- variables booléennes : $x_1, x_2, \ldots, x_n \in \{\textbf{vrai}, \textbf{faux}\}$.
- opérateurs :
  - $\neg X$: négation, inverse la valeur de X.
  - $X \wedge Y$: et, **vrai** ssi X et Y sont **vrai**.
  - $X \vee Y$: ou, **faux** ssi X et Y sont **faux**.

Définition

L'expression booléenne X est **satisfaisable** s'il existe une affectation de ses variables telle que X est **vrai**.

- $x_1 \wedge \neg x_2$ est satisfaite avec $x_1 = \textbf{vrai}$ et $x_2 = \textbf{faux}$.
- $\neg x_1 \wedge \neg (x_2 \vee x_3)$ est satisfaite avec $x_1 = x_2 = x_3 = \textbf{faux}$.

Le problème SAT

Problème (SAT)

L'expression booléenne X est-elle satisfaisable ?

Une expression booléenne est formée de :

- variables booléennes : $x_1, x_2, \ldots, x_n \in \{\textbf{vrai}, \textbf{faux}\}$.
- opérateurs :
  - $\neg X$: négation, inverse la valeur de X.
  - $X \wedge Y$: et, **vrai** ssi X et Y sont **vrai**.
  - $X \vee Y$: ou, **faux** ssi X et Y sont **faux**.

Définition

L'expression booléenne X est **satisfaisable** s'il existe une affectation de ses variables telle que X est **vrai**.

- $x_1 \wedge \neg x_2$ est satisfaite avec $x_1 = \textbf{vrai}$ et $x_2 = \textbf{faux}$.
- $\neg x_1 \wedge \neg (x_2 \vee x_3)$ est satisfaite avec $x_1 = x_2 = x_3 = \textbf{faux}$.
- $x_1 \wedge \neg x_1$ n'est pas satisfaisable.

Le problème SAT

Problème (SAT)

L'expression booléenne X est-elle satisfaisable ?

Une expression booléenne est formée de :

- variables booléennes : $x_1, x_2, \ldots, x_n \in \{\textbf{vrai}, \textbf{faux}\}$.
- opérateurs :
  - $\neg X$: négation, inverse la valeur de X.
  - $X \wedge Y$: et, **vrai** ssi X et Y sont **vrai**.
  - $X \vee Y$: ou, **faux** ssi X et Y sont **faux**.

Définition

L'expression booléenne X est **satisfaisable** s'il existe une affectation de ses variables telle que X est **vrai**.

- $x_1 \wedge \neg x_2$ est satisfaite avec $x_1 = \textbf{vrai}$ et $x_2 = \textbf{faux}$.
- $\neg x_1 \wedge \neg (x_2 \vee x_3)$ est satisfaite avec $x_1 = x_2 = x_3 = \textbf{faux}$.
- $x_1 \wedge \neg x_1$ n'est pas satisfaisable.

$$\text{SAT} \in \textbf{NP}$$

Définition (Forme normale conjonctive)

- Un **littéral** est une variable ou la négation d'une variable.
- Une **clause** est une disjonction de littéraux.
- Une expression booléenne sous **forme normale conjonctive** (FNC) est une conjonction de clauses.

Exemples :

- Littéraux : x_1, $\neg x_1$, x_2, $\neg x_2$, ...
- Clause : $(\neg x_1 \vee x_2)$, $(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4)$, (x_1).
- FNC : $(\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (x_1)$.

Définition (Forme normale conjonctive)

- Un **littéral** est une variable ou la négation d'une variable.
- Une **clause** est une disjonction de littéraux.
- Une expression booléenne sous **forme normale conjonctive** (FNC) est une conjonction de clauses.

Exemples :

- Littéraux : x_1, $\neg x_1$, x_2, $\neg x_2$, ...
- Clause : $(\neg x_1 \vee x_2)$, $(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4)$, (x_1).
- FNC : $(\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (x_1)$.

Une FNC est satisfaisable ssi il existe une affectation telle que dans chaque clause, au moins un littéral est **vrai**.

Définition

Une FNC est sous forme 3-FNC si toutes ses clauses contiennent exactement 3 littéraux.

- Pas FNC : $\neg(\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \wedge x_4) \wedge (x_1)$.
- FNC mais pas 3-FNC : $(\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (x_1)$.
- 3-FNC : $(\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_1 \vee x_1)$.

Définition

Une FNC est sous forme 3-FNC si toutes ses clauses contiennent exactement 3 littéraux.

- Pas FNC : $\neg(\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \wedge x_4) \wedge (x_1)$.
- FNC mais pas 3-FNC : $(\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (x_1)$.
- 3-FNC : $(\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_1 \vee x_1)$.

Toute expression booléenne peut être mise sous forme 3-FNC en un temps polynomial.

Le problème 3-SAT

Problème (3-SAT)

La 3-FNC X est-elle satisfaisable.

Le problème 3-SAT

Problème (3-SAT)

La 3-FNC X est-elle satisfaisable.

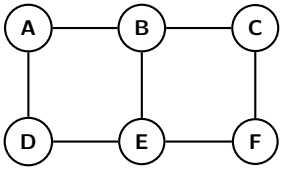
- 3-SAT $\preceq$ SAT : réduction par l'identité.
- SAT $\preceq$ 3-SAT : réduction par manipulations algébriques.

Le problème 3-couleur

Définition

Un **coloriage** de graphe est une attribution de couleurs à chacun de ses sommets de sorte que deux sommets adjacents aient de couleurs différentes.

Exemple :

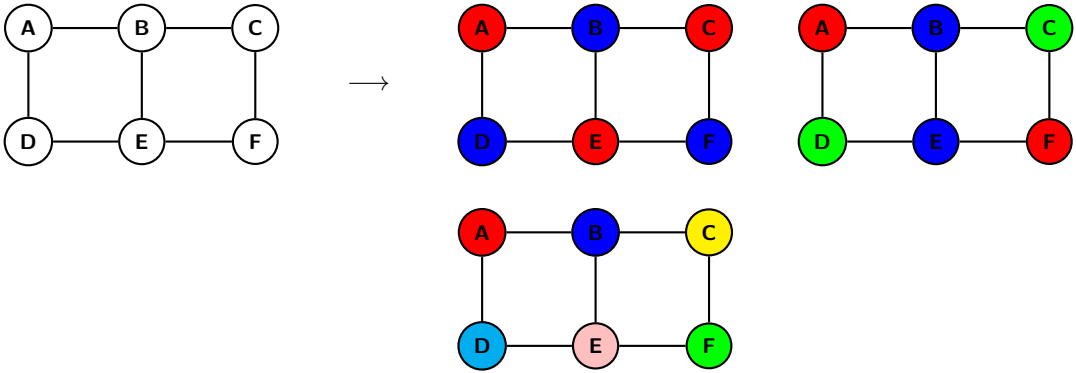


Le problème 3-couleur

Définition

Un **coloriage** de graphe est une attribution de couleurs à chacun de ses sommets de sorte que deux sommets adjacents aient de couleurs différentes.

Exemple :

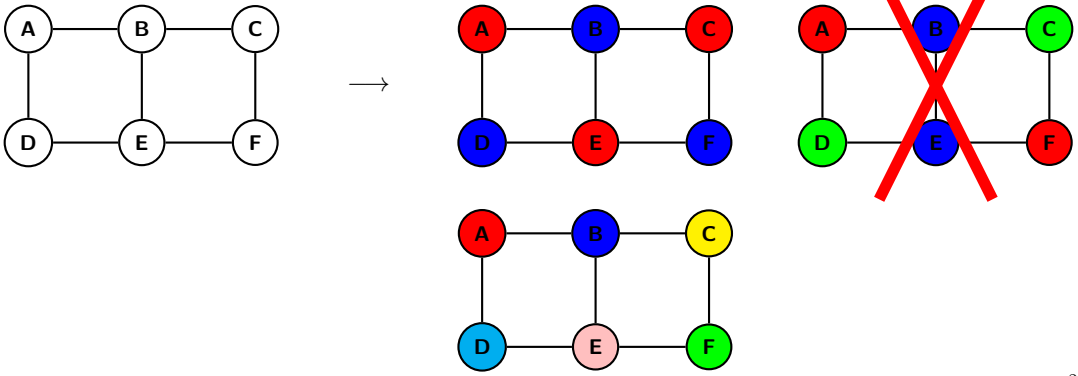


Le problème 3-couleur

Définition

Un **coloriage** de graphe est une attribution de couleurs à chacun de ses sommets de sorte que deux sommets adjacents aient de couleurs différentes.

Exemple :

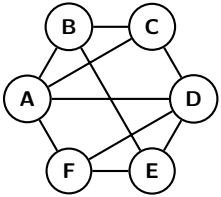


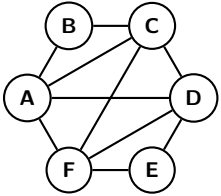
Les problèmes 3-couleurs

Problème

Étant donné un graphe G, existe-t-il un coloriage de G avec 3 couleurs.

Exemples :



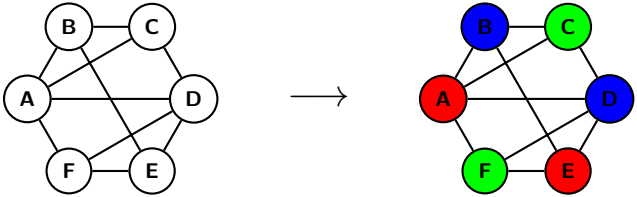


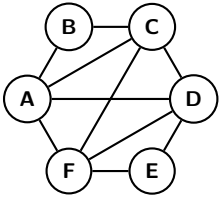
Les problèmes 3-couleurs

Problème

Étant donné un graphe G, existe-t-il un coloriage de G avec 3 couleurs.

Exemples :



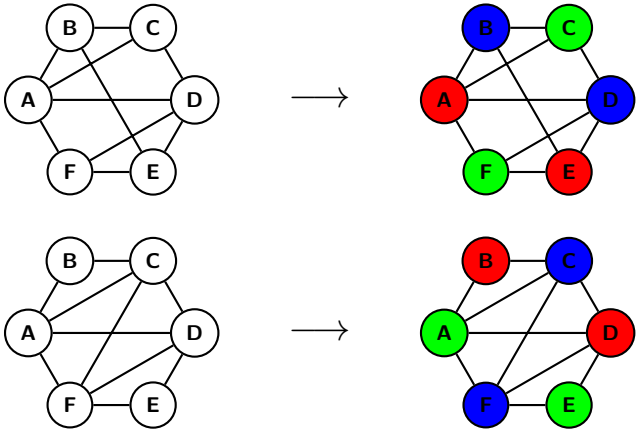


Les problèmes 3-couleurs

Problème

Étant donné un graphe G, existe-t-il un coloriage de G avec 3 couleurs.

Exemples :

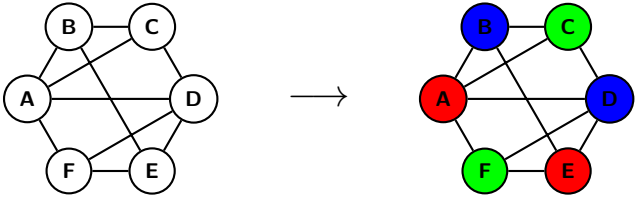


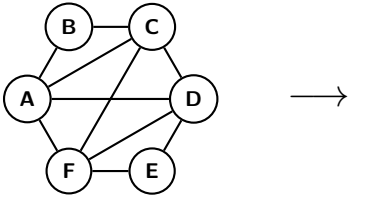
Les problèmes 3-couleurs

Problème

Étant donné un graphe G, existe-t-il un coloriage de G avec 3 couleurs.

Exemples :





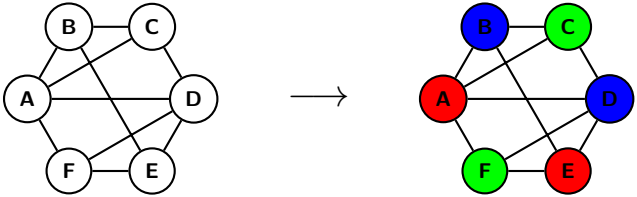
Il n'existe pas de coloriage avec seulement 3 couleurs.

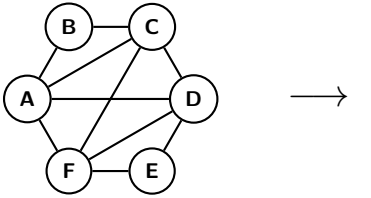
Les problèmes 3-couleurs

Problème

Étant donné un graphe G, existe-t-il un coloriage de G avec 3 couleurs.

Exemples :





Il n'existe pas de coloriage avec seulement 3 couleurs.

Le problème "*3-couleurs*" $\in$ **NP**.

Réduction de 3-SAT à 3-couleurs

Théorème

$$\text{3-couleurs} \preceq \quad \text{SAT}$$

Réduction de 3-SAT à 3-couleurs

Théorème

$$\text{3-couleurs} \preceq \text{3-SAT}$$

Réduction de 3-SAT à 3-couleurs

Théorème

$$\text{3-couleurs} \preceq \text{3-SAT} \preceq \text{3-couleurs}$$

Réduction de 3-SAT à 3-couleurs

Théorème

3-couleurs $\preceq$ 3-SAT $\preceq$ 3-couleurs

Exemple :

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_4)\,.$$

NP-complétude

Définition

Un problème Q est **NP-difficile** si pour tout problème $Q' \in \mathbf{NP}$, $\quad Q' \preceq Q$.

- Un algorithme pour Q peut être utilisé pour résoudre n'importe quel problème $Q' \in \mathbf{NP}$.

NP-complétude

Définition

Un problème Q est **NP-difficile** si pour tout problème $Q' \in \mathbf{NP}$, $\quad Q' \preceq Q$.

- Un algorithme pour Q peut être utilisé pour résoudre n'importe quel problème $Q' \in \mathbf{NP}$.
- Autrement dit, un problème **NP**-difficile *contient toute la difficulté de* **NP**.

Définition

Un problème Q est **NP-difficile** si pour tout problème $Q' \in \mathbf{NP}$, $\quad Q' \preceq Q$.

- Un algorithme pour Q peut être utilisé pour résoudre n'importe quel problème $Q' \in \mathbf{NP}$.
- Autrement dit, un problème **NP**-difficile *contient toute la difficulté de* **NP**.
- Si Q_1 est **NP**-difficile et $Q_1 \preceq Q_2$ alors Q_2 est **NP**-difficile.

NP-complétude

Définition

Un problème Q est **NP-difficile** si pour tout problème $Q' \in \mathbf{NP}$, $\quad Q' \preceq Q$.

- Un algorithme pour Q peut être utilisé pour résoudre n'importe quel problème $Q' \in \mathbf{NP}$.
- Autrement dit, un problème **NP**-difficile *contient toute la difficulté de* **NP**.
- Si Q_1 est **NP**-difficile et $Q_1 \preceq Q_2$ alors Q_2 est **NP**-difficile.

Définition

Un problème Q est **NP-complet** si

- $Q \in \mathbf{NP}$,
- Q est **NP**-difficile.

Théorème de Cook-Levin

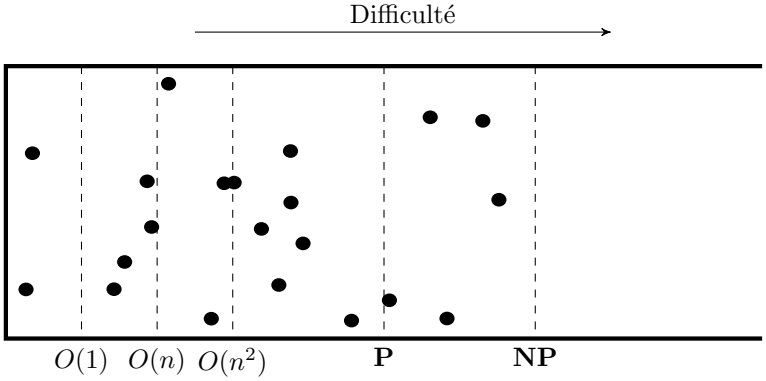
Théorème (Cook-Levin)

Le problème SAT est **NP**-complet.

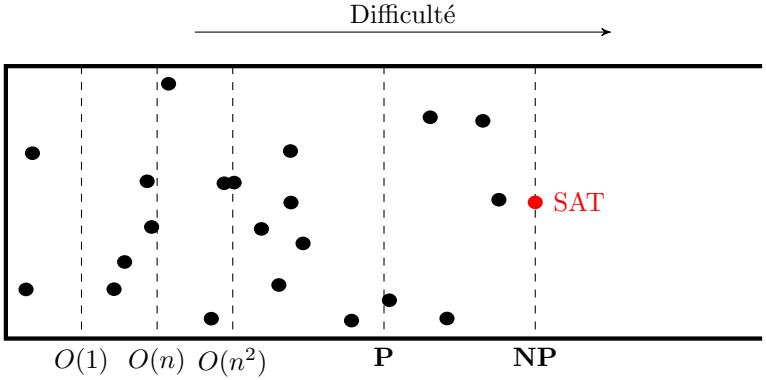
À montrer :

- SAT $\in$ **NP**.
- SAT est **NP**-difficile.

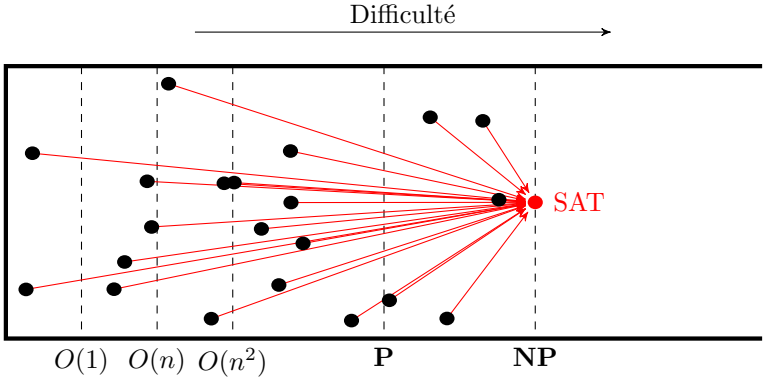
SAT est **NP**-difficile



SAT est **NP**-difficile

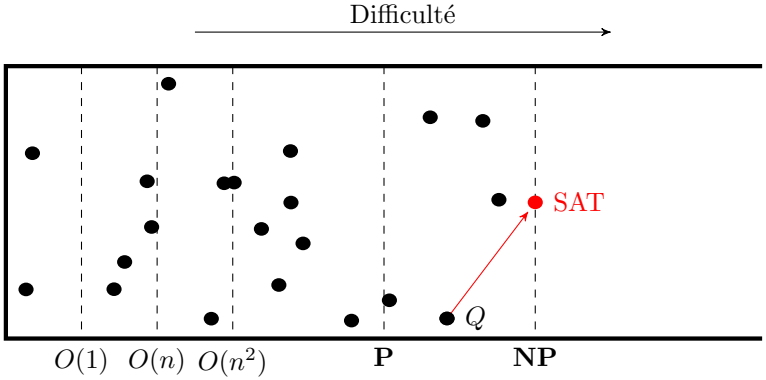


SAT est **NP**-difficile



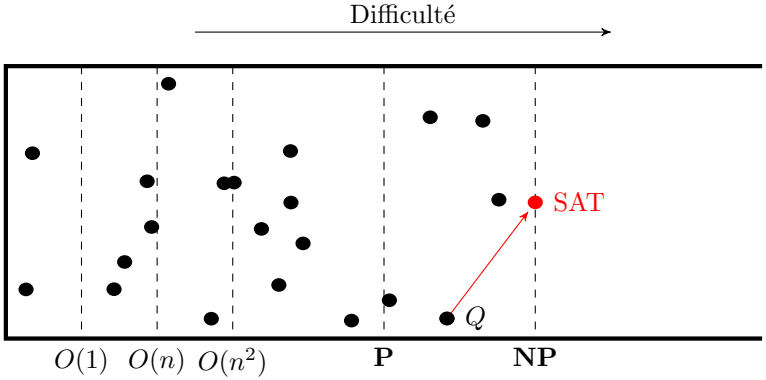
- $\forall Q \in \mathbf{NP}, \quad Q \preceq \text{SAT}.$

SAT est **NP**-difficile



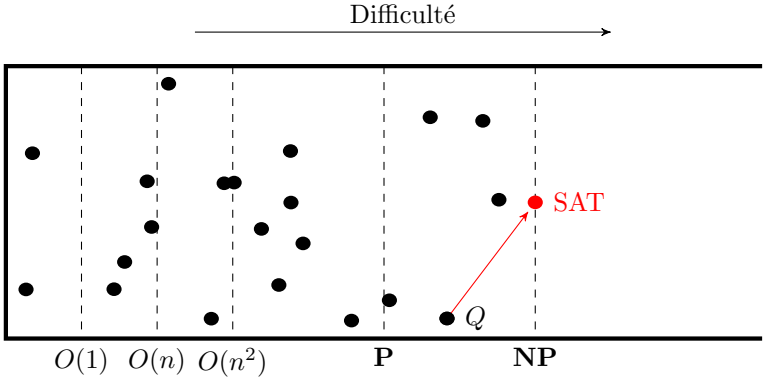
- $\forall Q \in \mathbf{NP}, \quad Q \preceq \text{SAT}$.
- Soit $Q \in \mathbf{NP}$.

SAT est **NP**-difficile



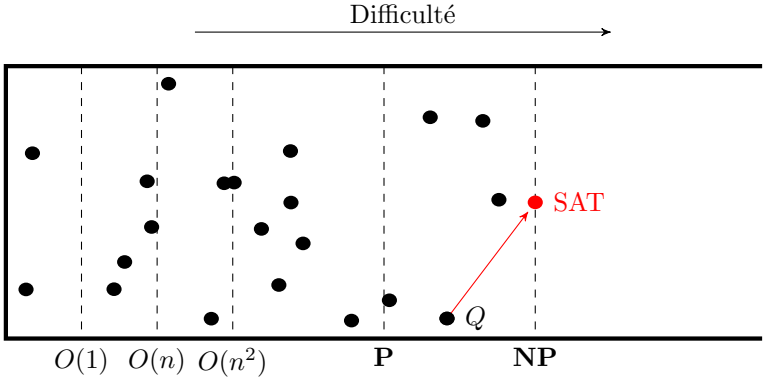
- $\forall Q \in \mathbf{NP}, \quad Q \preceq \text{SAT}$.
- Soit $Q \in \mathbf{NP}$.
- Il existe un algorithme de vérification en temps polynomial pour Q.

SAT est **NP**-difficile



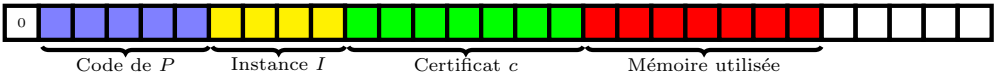
- $\forall Q \in \mathbf{NP}, \quad Q \preceq \text{SAT}$.
- Soit $Q \in \mathbf{NP}$.
- Il existe un algorithme de vérification en temps polynomial pour Q.
- Cet algorithme Q décrit un programme P pour le modèle RAM.

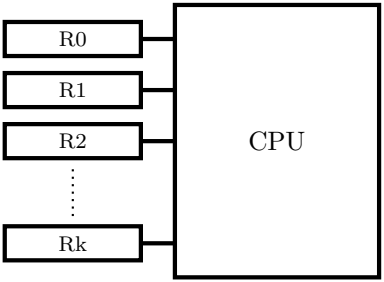
SAT est **NP**-difficile



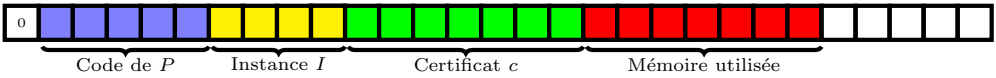
- $\forall Q \in \mathbf{NP}, \quad Q \preceq \text{SAT}$.
- Soit $Q \in \mathbf{NP}$.
- Il existe un algorithme de vérification en temps polynomial pour Q.
- Cet algorithme Q décrit un programme P pour le modèle RAM.
- Le temps d'exécution de P est borné par un certain polynôme $f(n)$.

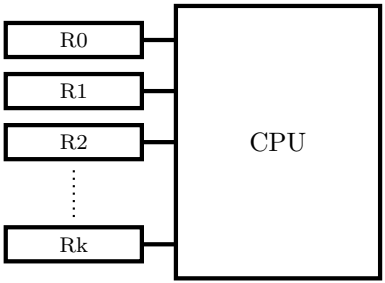
Encoder le modèle RAM





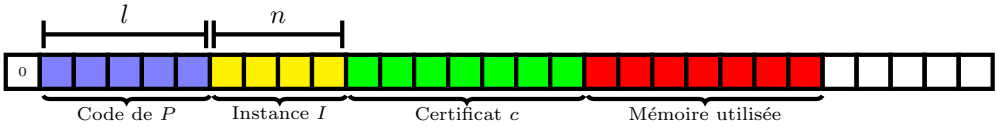
Encoder le modèle RAM

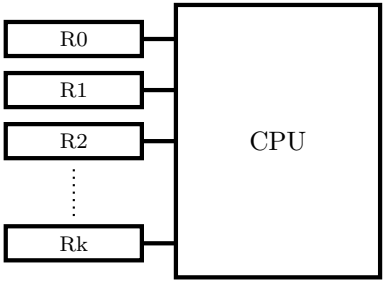




Nombre de cases mémoire utilisée :

Encoder le modèle RAM

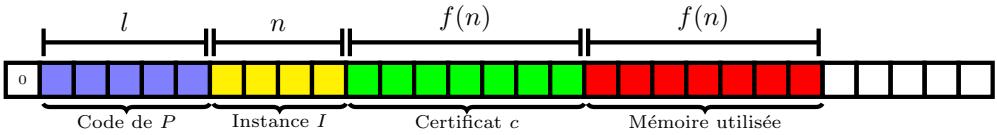


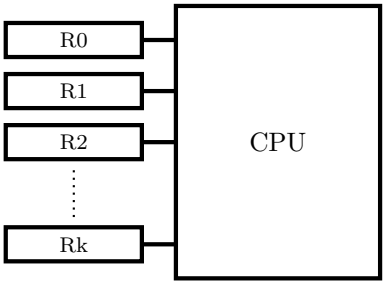


Nombre de cases mémoire utilisée :

- soit l la taille du programme,
- soit n la taille de l'instance I,

Encoder le modèle RAM

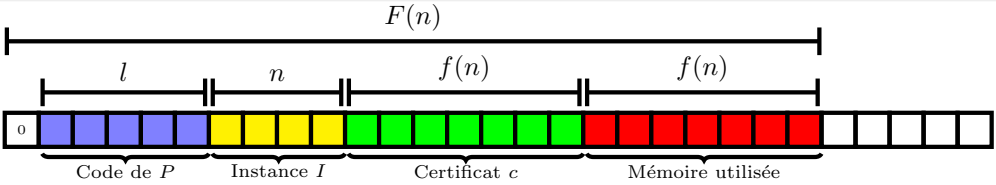


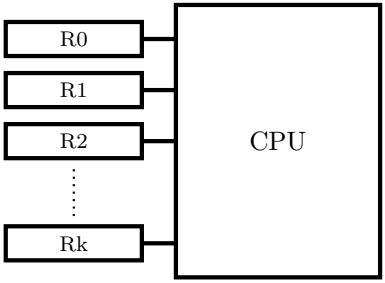


Nombre de cases mémoire utilisée :

- soit l la taille du programme,
- soit n la taille de l'instance I,
- la taille du certificat et la quantité de mémoire utilisée sont tous deux bornés par $f(n)$.

Encoder le modèle RAM

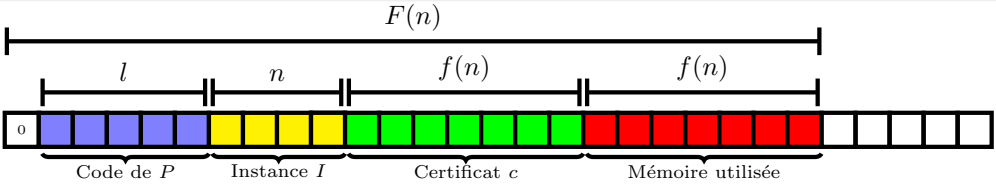


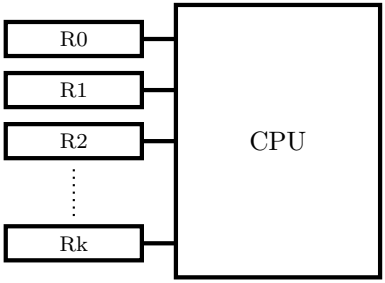


Nombre de cases mémoire utilisée :

- soit l la taille du programme,
- soit n la taille de l'instance I,
- la taille du certificat et la quantité de mémoire utilisée sont tous deux bornés par $f(n)$.
- On pose $F(n) = 1 + l + n + 2f(n)$.

Encoder le modèle RAM

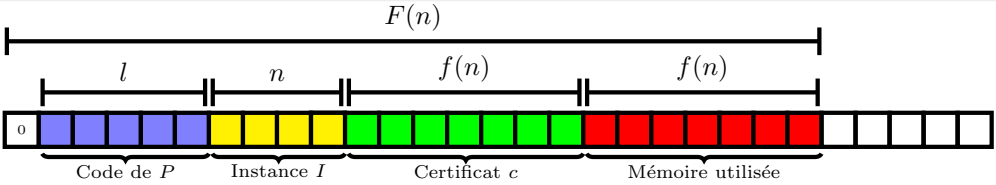


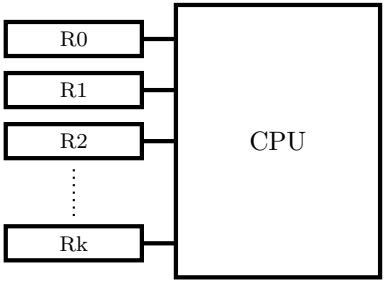


- Une expression booléenne E est construite par blocs :

$$E = B_1 \wedge B_2 \wedge B_3 \wedge \cdots \wedge B_{10}.$$

Encoder le modèle RAM



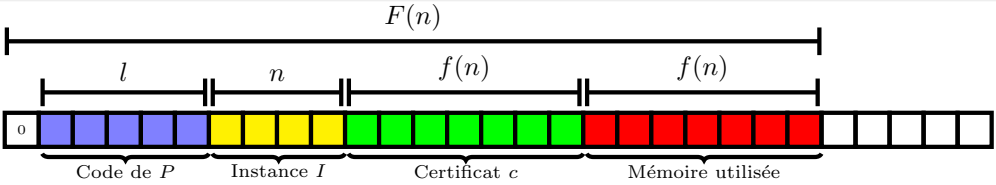


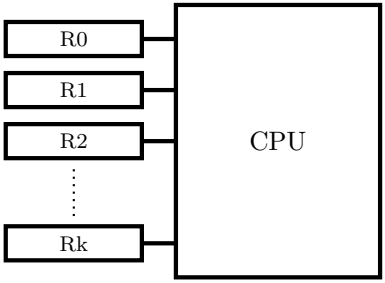
- Une expression booléenne E est construite par blocs :

$$E = B_1 \wedge B_2 \wedge B_3 \wedge \cdots \wedge B_{10}.$$

- Les variables sont :
  - $M^i_{j,x}$: à l'étape i, la case mémoire j contient la valeur x.
  - $R^i_{j,x}$: à l'étape i, le registe Rj contient la valeur x.

Encoder le modèle RAM

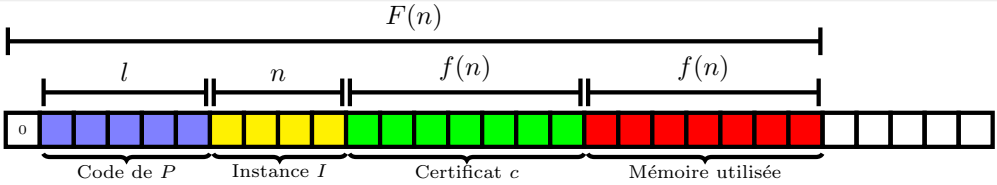


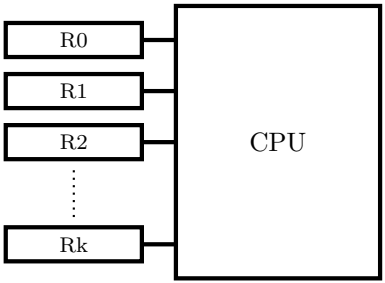


- À l'initialisation, la case mémoire 0 contient 0 :

$$B_1 = M^0_{0,0}$$

Encoder le modèle RAM





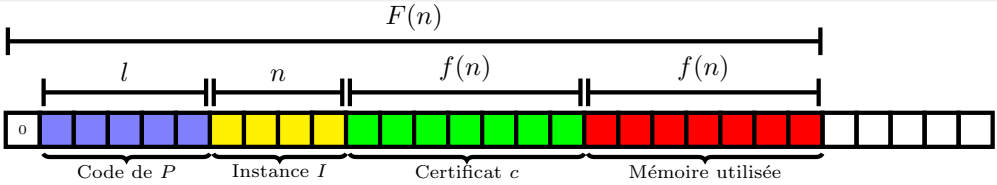
- À l'initialisation, la case mémoire 0 contient 0 :

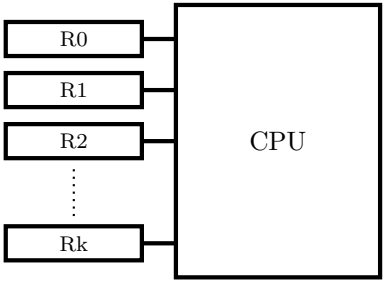
$$B_1 = M^0_{0,0}$$

- À l'initialisation, le registre R0 contient 1 :

$$B_2 = R^0_{0,1}$$

Encoder le modèle RAM

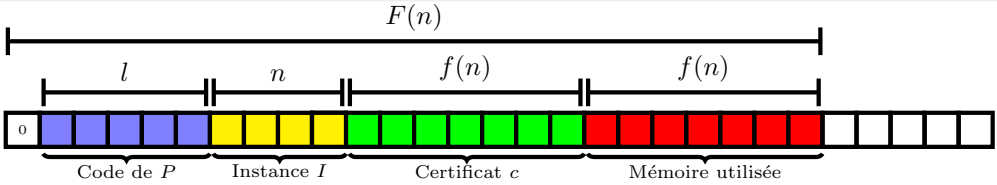


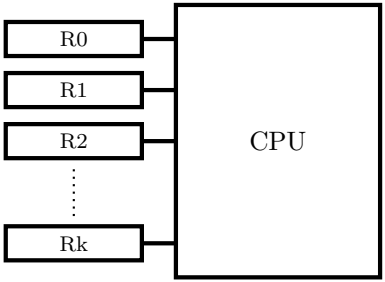


- À l'initialisation, les cases 1 à l contiennent le code du programme $C = c_1 c_2 \cdots c_l$:

$$B_3 = \bigwedge_{1 \leq i \leq l} M^0_{i,c_i}$$

Encoder le modèle RAM





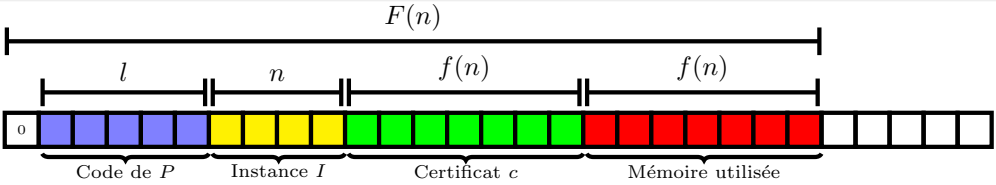
- À l'initialisation, les cases 1 à l contiennent le code du programme $C = c_1 c_2 \cdots c_l$:

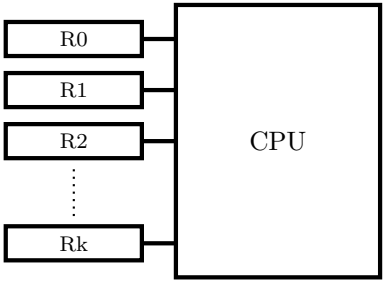
$$B_3 = \bigwedge_{1 \leq i \leq l} M^0_{i,c_i}$$

- À l'initialisation, les cases $l+1$ à $l+1+n$ contiennent le l'instance I :

$$B_4 = \bigwedge_{1 \leq i \leq l} M^0_{l+1+i,I_i}$$

Encoder le modèle RAM





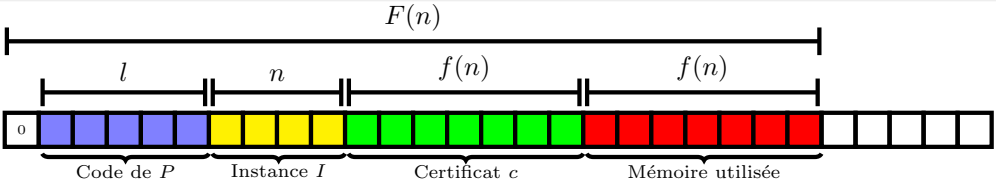
- À l'étape i, la case mémoire j contient une et une seule valeur :

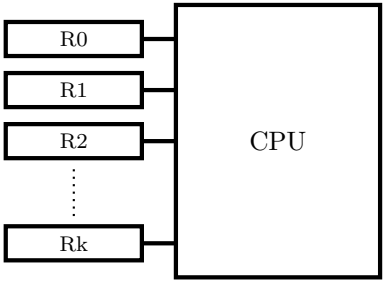
$$\alpha_j^i = \left(\bigvee_{0 \leq x < K} M_{j,x}^i \right) \bigwedge \left(\bigwedge_{0 \leq x < y < K} \left(\neg M_{j,x}^i \wedge \neg M_{j,y}^i \right) \right).$$

- À chaque étape, chacune des cases contiennent une et une seule valeur :

$$B_5 = \bigwedge_{\substack{0 \leq i \leq f(n) \\ 0 \leq j \leq F(n)}} \alpha_j^i.$$

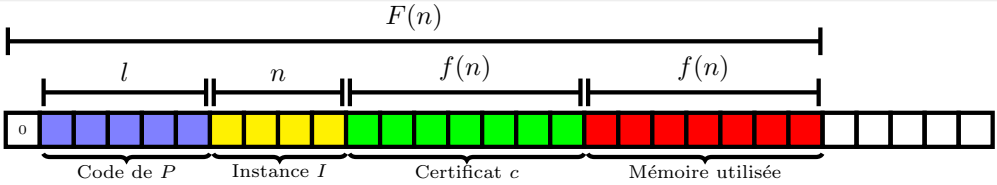
Encoder le modèle RAM

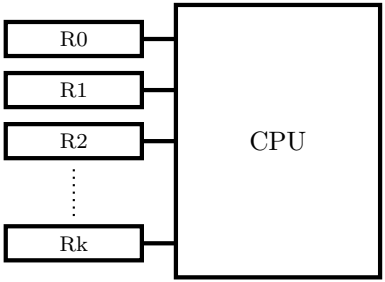




- À chaque étape, chacun des registres contient une et une seule valeur : $B_6 = \cdots$

Encoder le modèle RAM





- Opération $\texttt{Lire}(u, v)$:

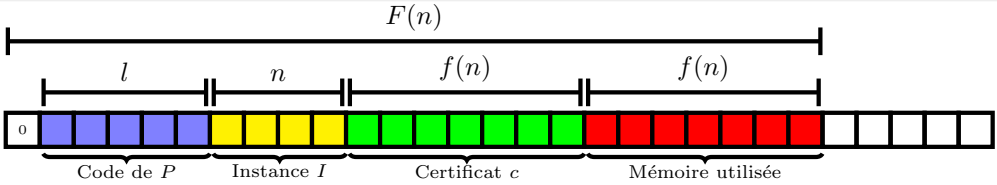
  Si à l'étape i, la prochaine instruction est $\texttt{Lire}(u, v)$ alors le contenu du registre Rv à l'étape $i+1$ est le même que le contenue de la case mémoire Ru à l'étape i.

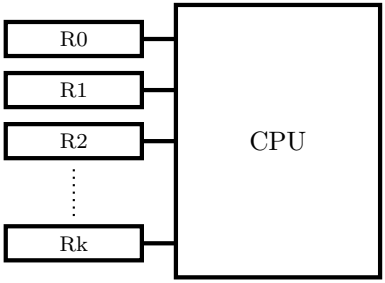
$$\beta^i_{u,v} = \bigwedge_{1 \le j \le l} \left(\neg R^i_{0,j} \vee \left(\neg M^i_{j,\texttt{Lire}(u,v)} \vee \left(\bigwedge_{0 \le j' < F(n)} \neg R^i_{u,j'} \vee \left(\bigwedge_{0 \le x < K} \neg M^i_{j',x} \vee R^{i+1}_{v,x} \right) \right) \right) \right)$$

- À chaque étape, pour chaque paire de registre :

$$B_7 = \bigwedge_{0 \le i \le f(n)} \left(\bigwedge_{0 \le u \le k} \left(\bigwedge_{0 \le v \le k} \beta^i_{u,v} \right) \right)$$

Encoder le modèle RAM

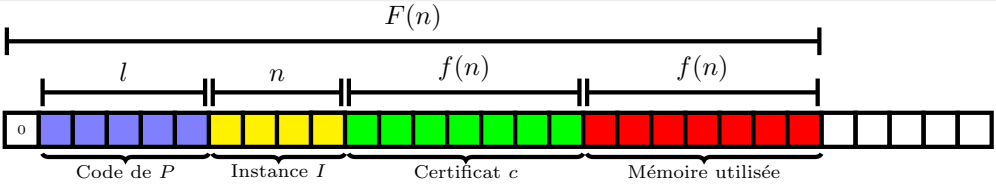


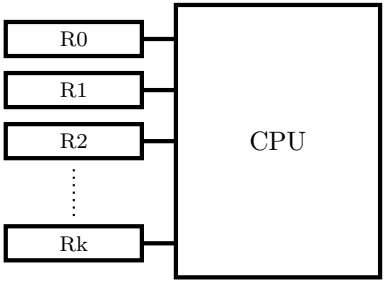


- Idem pour l'instruction $\texttt{Écrire}(u, v)$:

$$B_8 = \cdots$$

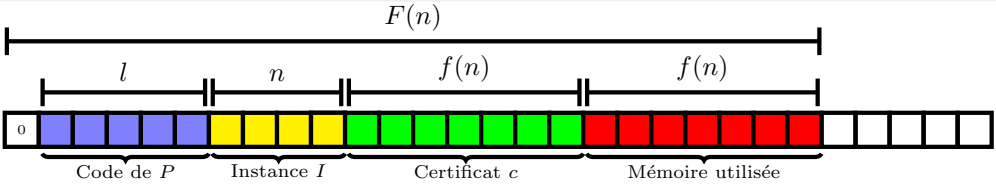
Encoder le modèle RAM

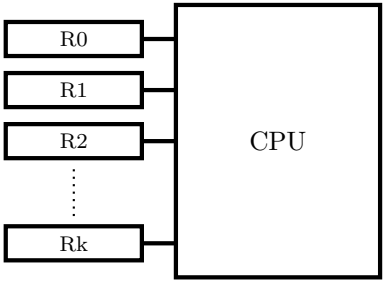




- Idem pour l'instruction $\texttt{Écrire}(u, v)$:
$$B_8 = \cdots$$
- Les opérations sur les registres :
$$B_9 = \cdots$$

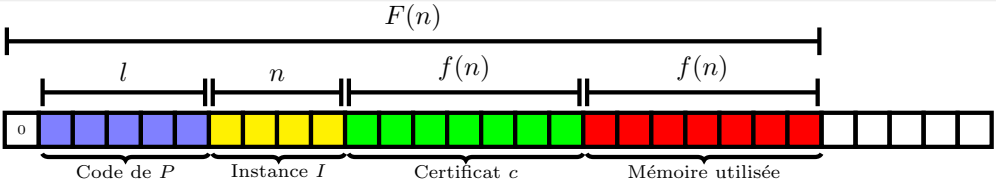
Encoder le modèle RAM

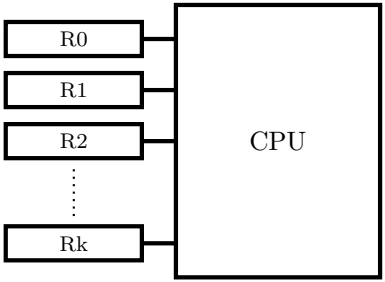




- Idem pour l'instruction $\texttt{Écrire}(u, v)$:
$$B_8 = \cdots$$
- Les opérations sur les registres :
$$B_9 = \cdots$$
- À la fin de l'exécution, la case mémoire 0 contient la valeur 1 :
$$B_{10} = M_{0,1}^{f(n)}$$

Encoder le modèle RAM

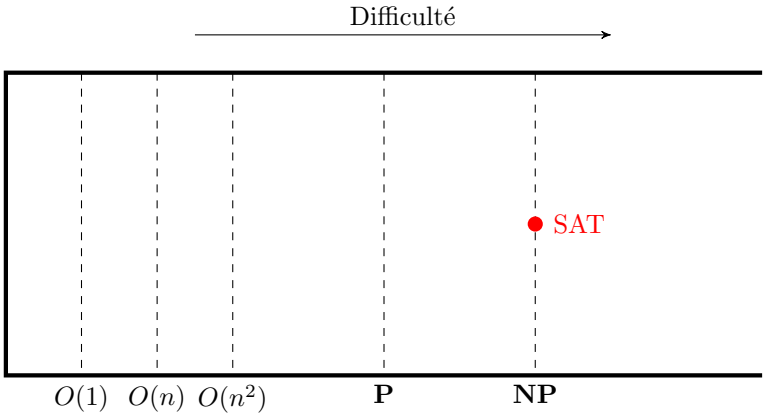




- Idem pour l'instruction $\texttt{Écrire}(u, v)$:
$$B_8 = \cdots$$
- Les opérations sur les registres :
$$B_9 = \cdots$$
- À la fin de l'exécution, la case mémoire 0 contient la valeur 1 :
$$B_{10} = M_{0,1}^{f(n)}$$
- E est satisfaisable ssi le programme accepte l'instance I.

□

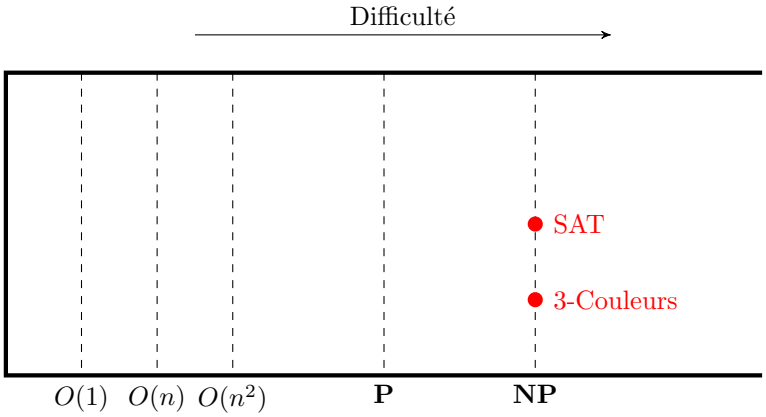
Problèmes **NP**-complets



On a vu que :

- 3-Couleurs $\in$ **NP**.

Problèmes **NP**-complets



On a vu que :

- 3-Couleurs $\in$ **NP**.
- SAT $\preceq$ 3-SAT $\preceq$ 3-couleurs.
- Ainsi, 3-couleurs est **NP**-complet.

Problèmes **NP**-complets

- Pour montrer qu'un problème Q est **NP**-complet, il faut :
  - fournir un algorithme de vérification en temps polynomial,
  - montrer qu'il existe un problème **NP**-complet Q' tel que $Q' \preceq Q$.

Problèmes **NP**-complets

- Pour montrer qu'un problème Q est **NP**-complet, il faut :
  - fournir un algorithme de vérification en temps polynomial,
  - montrer qu'il existe un problème **NP**-complet Q' tel que $Q' \preceq Q$.
- En 1972 Karp fournir une liste de 21 problèmes **NP**-complets. (lien vers le pdf)

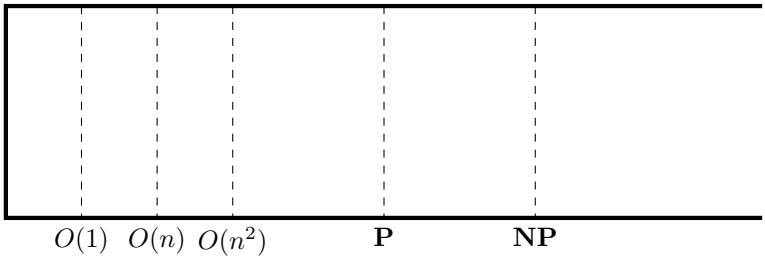
Problèmes **NP**-complets

- Pour montrer qu'un problème Q est **NP**-complet, il faut :
  - fournir un algorithme de vérification en temps polynomial,
  - montrer qu'il existe un problème **NP**-complet Q' tel que $Q' \preceq Q$.
- En 1972 Karp fournir une liste de 21 problèmes **NP**-complets. (lien vers le pdf)
- Aujourd'hui on en connaît des centaines. (lien vers Wikipedia)

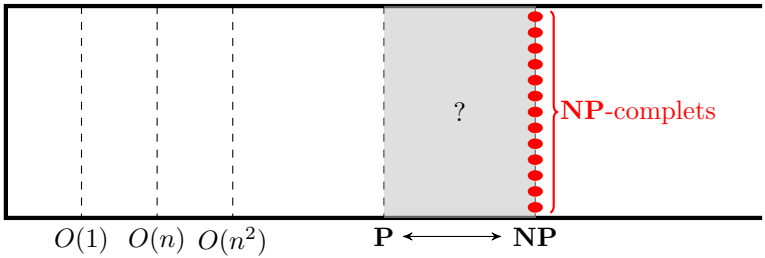
Problèmes **NP**-complets

- Pour montrer qu'un problème Q est **NP**-complet, il faut :
  - fournir un algorithme de vérification en temps polynomial,
  - montrer qu'il existe un problème **NP**-complet Q' tel que $Q' \preceq Q$.
- En 1972 Karp fournir une liste de 21 problèmes **NP**-complets. (lien vers le pdf)
- Aujourd'hui on en connaît des centaines. (lien vers Wikipedia)
- Exemples :
  - programmation linéaire en nombre entiers,
  - partitionner un ensemble de nombres en deux sous-ensembles de mêmes sommes,
  - circuit hamiltonien,
  - commis voyageur,
  - les jeux : Battleship, Lemmings, Tetris*, Rubik's cube*, ...

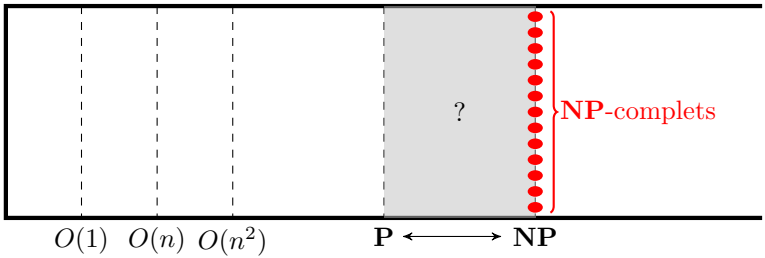
Conclusion



Conclusion

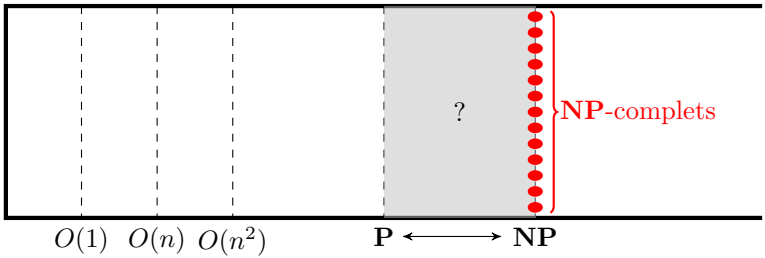


Conclusion



- Pour prouver **P** = **NP** :
  - Il suffit de trouver un algorithme polynomial pour résoudre **un** problème **NP**-complet.

Conclusion



- Pour prouver **P** = **NP** :
  - Il suffit de trouver un algorithme polynomial pour résoudre **un** problème **NP**-complet.
- Pour prouver **P** ≠ **NP** :
  - Il suffit de montrer qu'**un** problème **NP**-complet qui ne peut pas être résolu en temps polynomial.