Complexité des algorithmes probabilistes

Sylvain Perifel

LIP, ENS Lyon

Ski, le 21 janvier 2008

Introduction

- Utilisation de bits aléatoires : accélération de certains algorithmes. Vraie utilité en pratique, bits pseudo-aléatoires.

Introduction

- Utilisation de bits aléatoires : accélération de certains algorithmes. Vraie utilité en pratique, bits pseudo-aléatoires.

- Y a-t-il une accélération exponentielle ? super-polynomiale ?

Introduction

- Utilisation de bits aléatoires : accélération de certains algorithmes. Vraie utilité en pratique, bits pseudo-aléatoires.
- Y a-t-il une accélération exponentielle ? super-polynomiale ?
- Petit à petit les algorithmes sont dérandomisés, et
- on obtient des indices suggérant qu'ils peuvent tous être dérandomisés.

Introduction

- Utilisation de bits aléatoires : accélération de certains algorithmes. Vraie utilité en pratique, bits pseudo-aléatoires.
- Y a-t-il une accélération exponentielle ? super-polynomiale ?
- Petit à petit les algorithmes sont dérandomisés, et
- on obtient des indices suggérant qu'ils peuvent tous être dérandomisés.

Dérandomisation : de la pratique à la théorie.

Plan

Plan

Quick sort

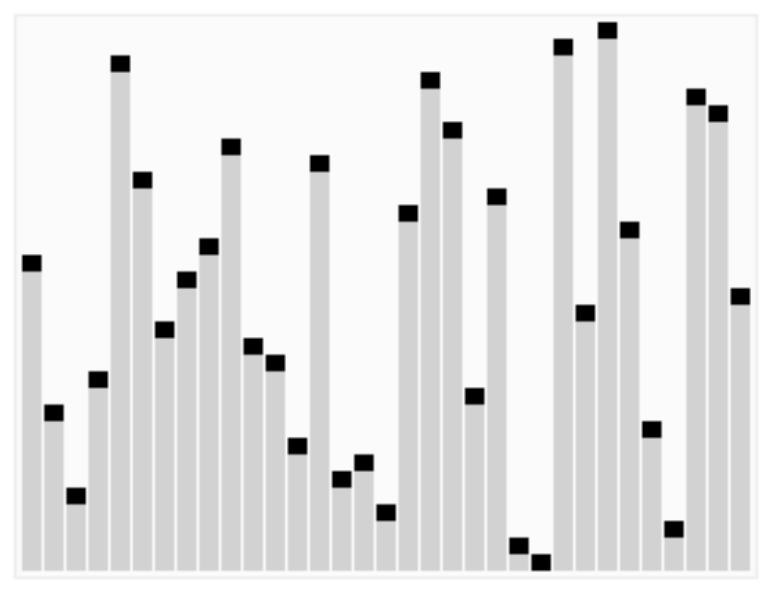
Quick sort déterministe

1. On choisit le dernier élément de la liste comme pivot.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.

Quick sort

Quick sort déterministe

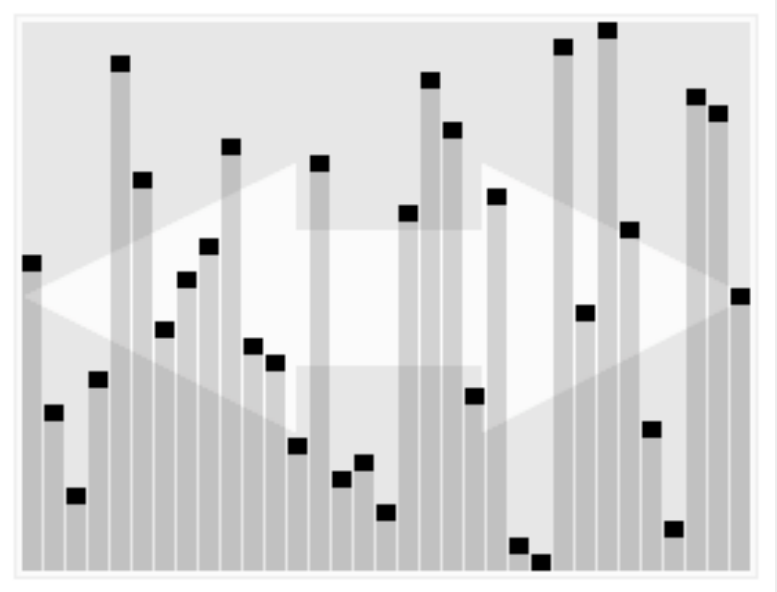
1. On choisit le dernier élément de la liste comme pivot.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.



Quick sort

Quick sort déterministe

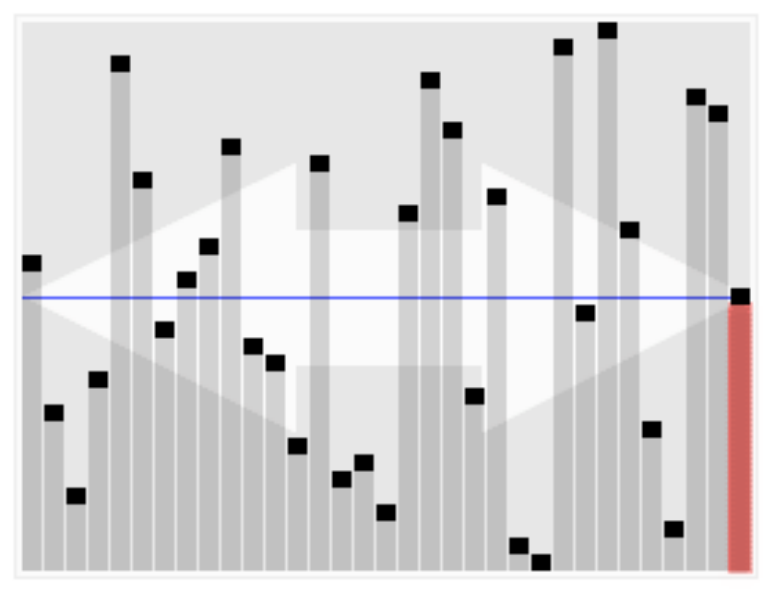
1. On choisit le dernier élément de la liste comme pivot.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.



Quick sort

Quick sort déterministe

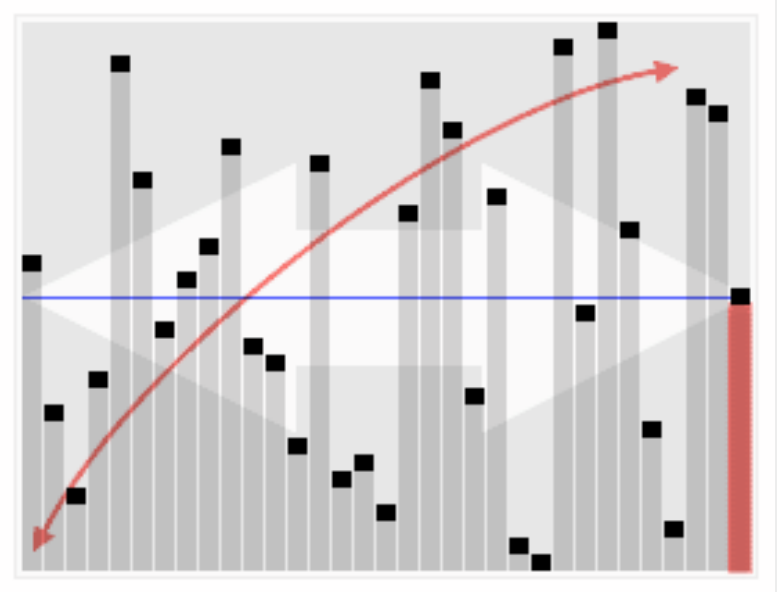
1. On choisit le dernier élément de la liste comme pivot.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.



Quick sort

Quick sort déterministe

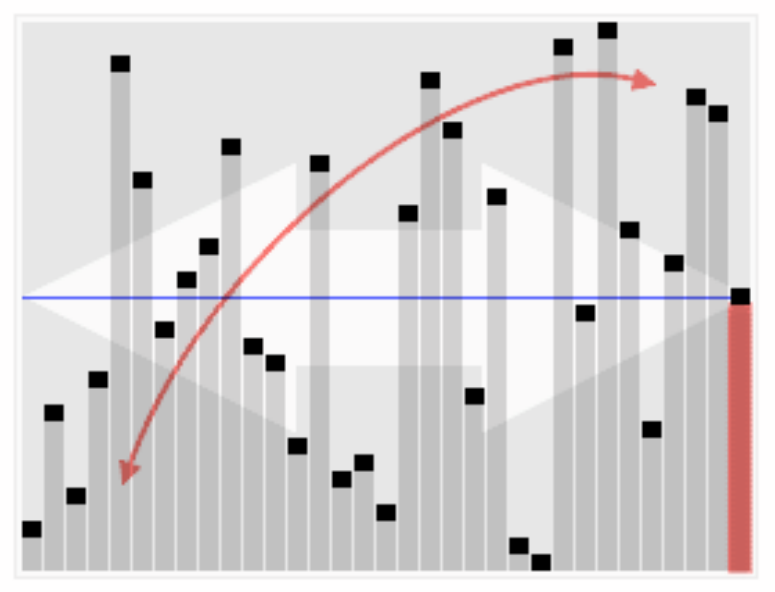
1. On choisit le dernier élément de la liste comme pivot.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.



Quick sort

Quick sort déterministe

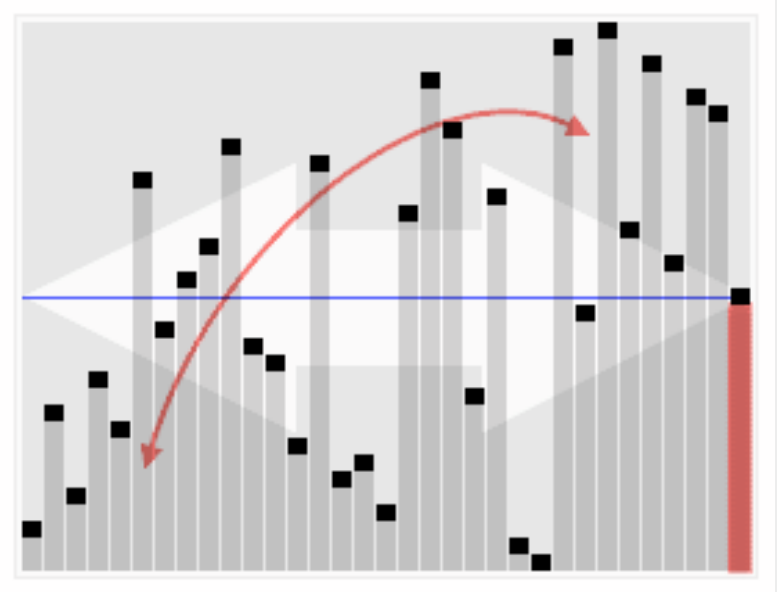
1. On choisit le dernier élément de la liste comme pivot.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.



Quick sort

Quick sort déterministe

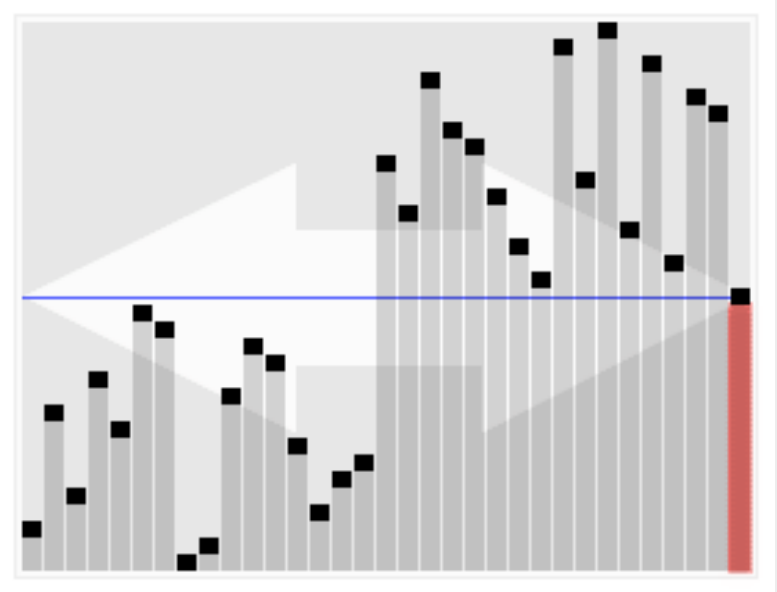
1. On choisit le dernier élément de la liste comme pivot.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.



Quick sort

Quick sort déterministe

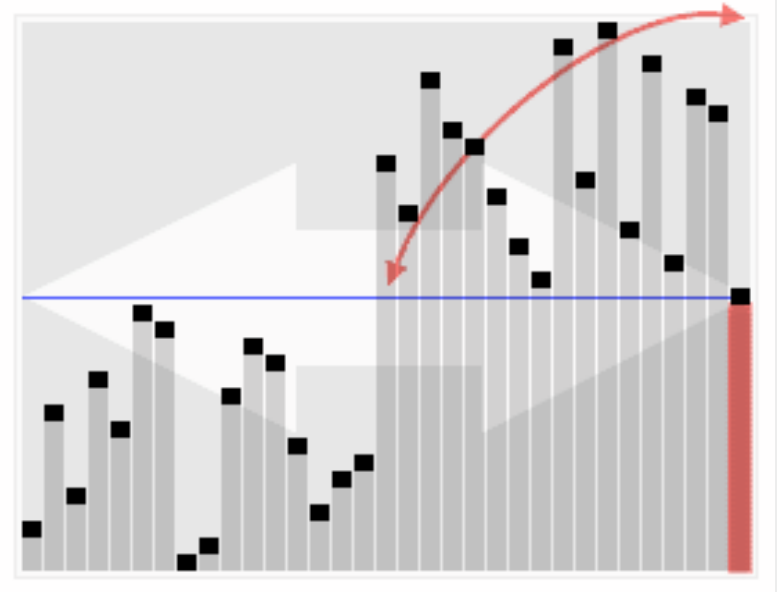
1. On choisit le dernier élément de la liste comme pivot.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.



Quick sort

Quick sort déterministe

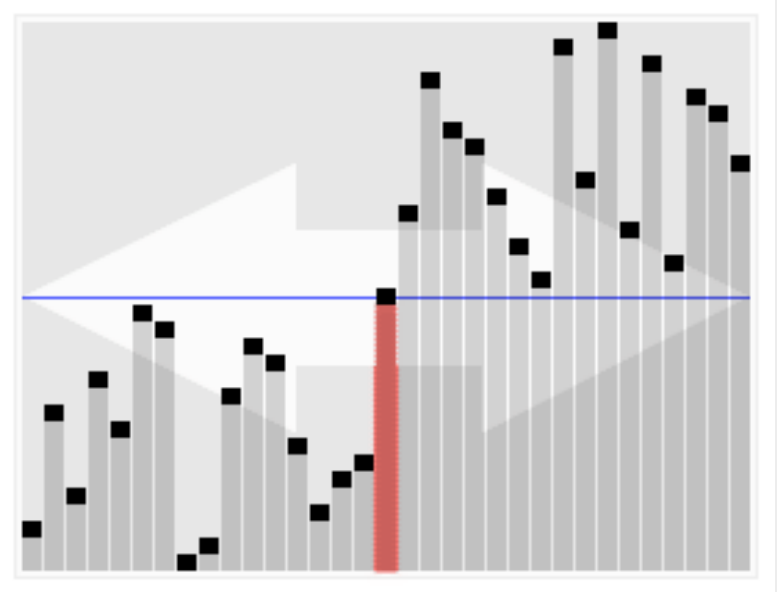
1. On choisit le dernier élément de la liste comme pivot.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.



Quick sort

Quick sort déterministe

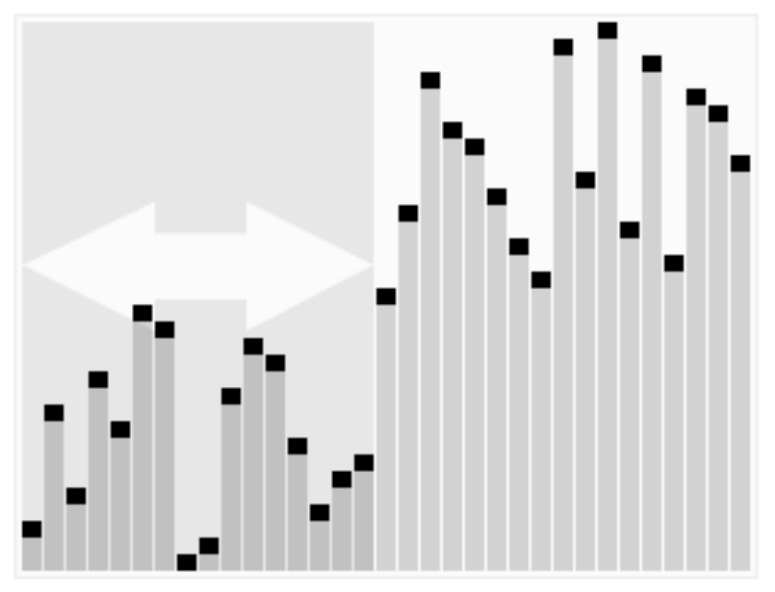
1. On choisit le dernier élément de la liste comme pivot.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.



Quick sort

Quick sort déterministe

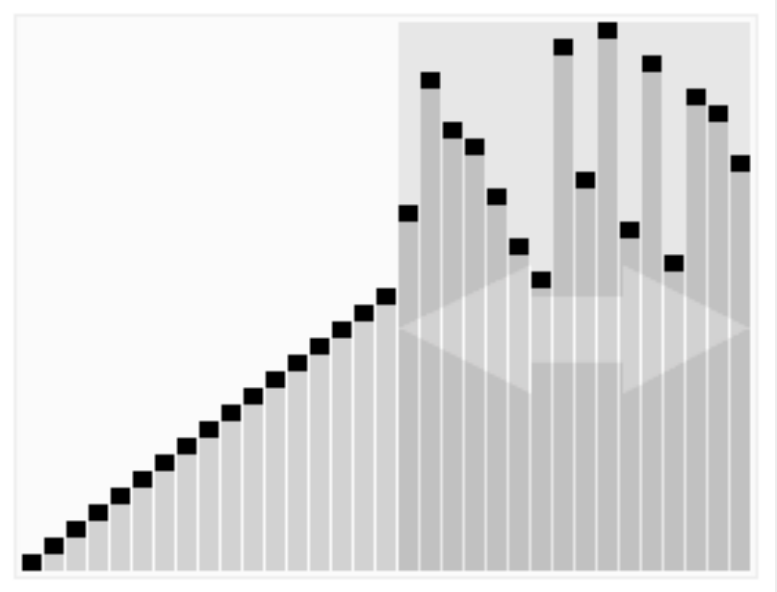
1. On choisit le dernier élément de la liste comme pivot.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.



Quick sort

Quick sort déterministe

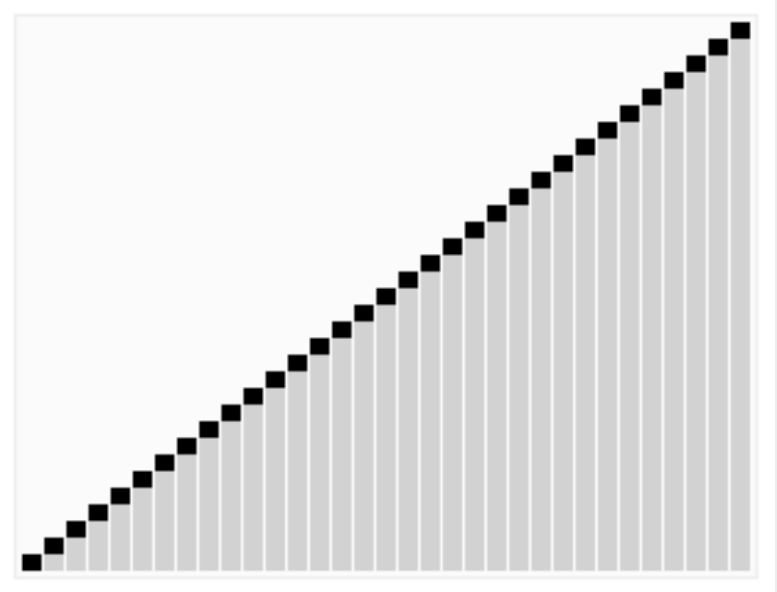
1. On choisit le dernier élément de la liste comme pivot.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.



Quick sort

Quick sort déterministe

1. On choisit le dernier élément de la liste comme pivot.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.



Quick sort

Quick sort déterministe

1. On choisit le dernier élément de la liste comme pivot.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.

Complexité de l'algorithme : $O(n \log n)$ en moyenne, $O(n^2)$ dans le pire cas.

Quick sort

Quick sort probabiliste

1. On choisit le pivot aléatoirement.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.

Complexité de l'algorithme : $O(n \log n)$ en moyenne, $O(n^2)$ dans le pire cas.

Version probabiliste

Quick sort

Quick sort probabiliste

1. On choisit le pivot aléatoirement.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.

Complexité de l'algorithme : $O(n \log n)$ en moyenne, $O(n^2)$ dans le pire cas.

Version probabiliste : espérance du temps d'exécution $O(n \log n)$ quelque soit l'instance.

Quick sort

Quick sort probabiliste

1. On choisit le pivot aléatoirement.
2. On place tous les éléments plus petits avant et tous les plus grands après.
3. On trie récursivement les deux parties de la liste.

Complexité de l'algorithme : $O(n \log n)$ en moyenne, $O(n^2)$ dans le pire cas.

Version probabiliste : espérance du temps d'exécution $O(n \log n)$ quelque soit l'instance. Récurrence :

$$T(n) = \alpha n + \frac{1}{n} \sum_{k=0}^{n-1} (T(k) + T(n-k)).$$

Médiane (1/3)

Problème Sélection

Entrée : une liste de n entiers distincts, et un entier i.

Sortie : le i-ème plus grand élément de la liste.

Médiane (1/3)

Problème Sélection

Entrée : une liste de n entiers distincts, et un entier i.

Sortie : le i-ème plus grand élément de la liste.

Remarque : trouver la médiane est un cas particulier.

Médiane (1/3)

Problème Sélection
Entrée : une liste de n entiers distincts, et un entier i.
Sortie : le i-ème plus grand élément de la liste.

Remarque : trouver la médiane est un cas particulier.

- Algorithme déterministe évident : $O(n \log n)$ en triant.

Médiane (1/3)

Problème Sélection
Entrée : une liste de n entiers distincts, et un entier i.
Sortie : le i-ème plus grand élément de la liste.

Remarque : trouver la médiane est un cas particulier.

- Algorithme déterministe évident : $O(n \log n)$ en triant.
- Algorithme déterministe compliqué en $O(n)$.

Médiane (1/3)

Problème Sélection
Entrée : une liste de n entiers distincts, et un entier i.
Sortie : le i-ème plus grand élément de la liste.

Remarque : trouver la médiane est un cas particulier.

- Algorithme déterministe évident : $O(n \log n)$ en triant.
- Algorithme déterministe compliqué en $O(n)$.
- Algorithme probabiliste simple en $O(n)$.

Médiane (2/3)

Algorithme probabiliste :

1. Choisir un pivot p au hasard.
2. Partitionner la liste en deux sous-listes : L_1 contenant les éléments inférieurs à p et L_2 les éléments supérieurs.
3. En comptant le nombre d'éléments de L_1, déterminer si l'élément recherché est dans L_1 ou L_2.
4. Continuer la recherche récursivement dans la bonne sous-liste.

Médiane (2/3)

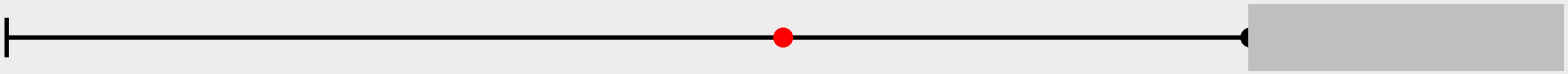
Algorithme probabiliste :

1. Choisir un pivot p au hasard.
2. Partitionner la liste en deux sous-listes : L_1 contenant les éléments inférieurs à p et L_2 les éléments supérieurs.
3. En comptant le nombre d'éléments de L_1, déterminer si l'élément recherché est dans L_1 ou L_2.
4. Continuer la recherche récursivement dans la bonne sous-liste.

Médiane (2/3)

Algorithme probabiliste :

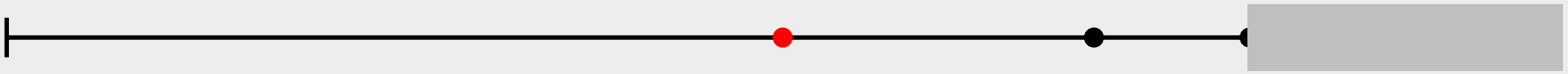
1. Choisir un pivot p au hasard.
2. Partitionner la liste en deux sous-listes : L_1 contenant les éléments inférieurs à p et L_2 les éléments supérieurs.
3. En comptant le nombre d'éléments de L_1, déterminer si l'élément recherché est dans L_1 ou L_2.
4. Continuer la recherche récursivement dans la bonne sous-liste.



Médiane (2/3)

Algorithme probabiliste :

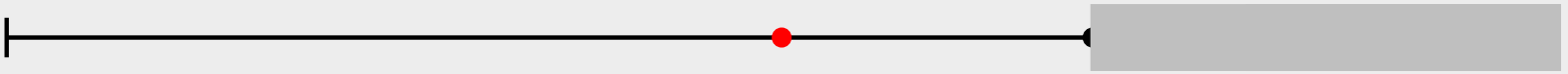
1. Choisir un pivot p au hasard.
2. Partitionner la liste en deux sous-listes : L_1 contenant les éléments inférieurs à p et L_2 les éléments supérieurs.
3. En comptant le nombre d'éléments de L_1, déterminer si l'élément recherché est dans L_1 ou L_2.
4. Continuer la recherche récursivement dans la bonne sous-liste.



Médiane (2/3)

Algorithme probabiliste :

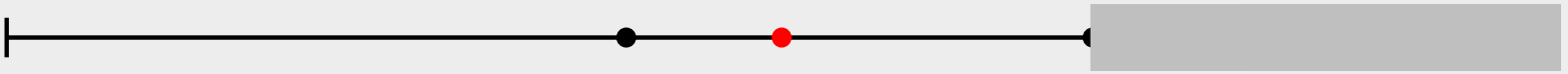
1. Choisir un pivot p au hasard.
2. Partitionner la liste en deux sous-listes : L_1 contenant les éléments inférieurs à p et L_2 les éléments supérieurs.
3. En comptant le nombre d'éléments de L_1, déterminer si l'élément recherché est dans L_1 ou L_2.
4. Continuer la recherche récursivement dans la bonne sous-liste.



Médiane (2/3)

Algorithme probabiliste :

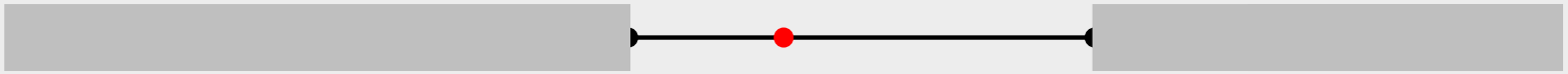
1. Choisir un pivot p au hasard.
2. Partitionner la liste en deux sous-listes : L_1 contenant les éléments inférieurs à p et L_2 les éléments supérieurs.
3. En comptant le nombre d'éléments de L_1, déterminer si l'élément recherché est dans L_1 ou L_2.
4. Continuer la recherche récursivement dans la bonne sous-liste.



Médiane (2/3)

Algorithme probabiliste :

1. Choisir un pivot p au hasard.
2. Partitionner la liste en deux sous-listes : L_1 contenant les éléments inférieurs à p et L_2 les éléments supérieurs.
3. En comptant le nombre d'éléments de L_1, déterminer si l'élément recherché est dans L_1 ou L_2.
4. Continuer la recherche récursivement dans la bonne sous-liste.



Médiane (2/3)

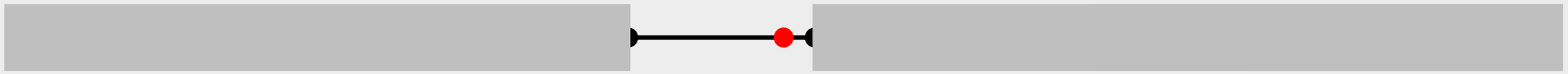
Algorithme probabiliste :

1. Choisir un pivot p au hasard.
2. Partitionner la liste en deux sous-listes : L_1 contenant les éléments inférieurs à p et L_2 les éléments supérieurs.
3. En comptant le nombre d'éléments de L_1, déterminer si l'élément recherché est dans L_1 ou L_2.
4. Continuer la recherche récursivement dans la bonne sous-liste.

Médiane (2/3)

Algorithme probabiliste :

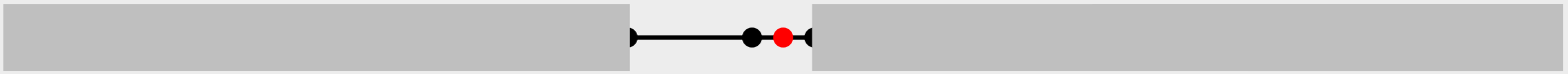
1. Choisir un pivot p au hasard.
2. Partitionner la liste en deux sous-listes : L_1 contenant les éléments inférieurs à p et L_2 les éléments supérieurs.
3. En comptant le nombre d'éléments de L_1, déterminer si l'élément recherché est dans L_1 ou L_2.
4. Continuer la recherche récursivement dans la bonne sous-liste.



Médiane (2/3)

Algorithme probabiliste :

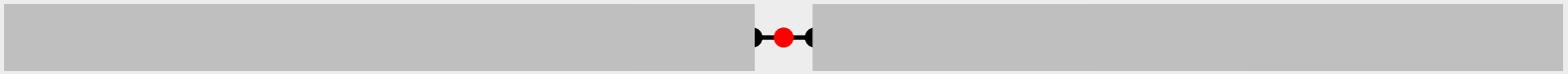
1. Choisir un pivot p au hasard.
2. Partitionner la liste en deux sous-listes : L_1 contenant les éléments inférieurs à p et L_2 les éléments supérieurs.
3. En comptant le nombre d'éléments de L_1, déterminer si l'élément recherché est dans L_1 ou L_2.
4. Continuer la recherche récursivement dans la bonne sous-liste.



Médiane (2/3)

Algorithme probabiliste :

1. Choisir un pivot p au hasard.
2. Partitionner la liste en deux sous-listes : L_1 contenant les éléments inférieurs à p et L_2 les éléments supérieurs.
3. En comptant le nombre d'éléments de L_1, déterminer si l'élément recherché est dans L_1 ou L_2.
4. Continuer la recherche récursivement dans la bonne sous-liste.



Médiane (3/3)

Analyse de l'algorithme : espérance du temps d'exécution.

$$T(n) = n + \frac{2}{n} \sum_{k=n/2}^{n-1} T(k).$$

Médiane (3/3)

Analyse de l'algorithme : espérance du temps d'exécution.

$$T(n) = n + \frac{2}{n} \sum_{k=n/2}^{n-1} T(k).$$

Par récurrence, $T(k) \leq \alpha k$ donc

$$T(n) \leq n + \frac{2\alpha}{n} \sum_{k/2}^{n-1} k$$

Médiane (3/3)

Analyse de l'algorithme : espérance du temps d'exécution.

$$T(n) = n + \frac{2}{n} \sum_{k=n/2}^{n-1} T(k).$$

Par récurrence, $T(k) \leq \alpha k$ donc

$$T(n) \leq n + \frac{2\alpha}{n} \sum_{k/2}^{n-1} k \leq n + \frac{2\alpha}{n}(\frac{n^2}{2} - \frac{n^2}{8})$$

Médiane (3/3)

Analyse de l'algorithme : espérance du temps d'exécution.

$$T(n) = n + \frac{2}{n} \sum_{k=n/2}^{n-1} T(k).$$

Par récurrence, $T(k) \leq \alpha k$ donc

$$T(n) \leq n + \frac{2\alpha}{n} \sum_{k/2}^{n-1} k \leq n + \frac{2\alpha}{n}(\frac{n^2}{2} - \frac{n^2}{8})$$

$$\leq (1 + \frac{3}{4}\alpha)n \leq \alpha n \text{ si } \alpha \geq 4.$$

Nombres premiers (1/3)

Problème : déterminer si un nombre n donné en entrée est premier.

Nombres premiers (1/3)

Problème : déterminer si un nombre n donné en entrée est premier.

- Algorithme probabiliste polynomial de Rabin-Miller, 1976.
- Algorithme déterministe polynomial de Agrawal-Kayal-Saxena, 2002.

Nombres premiers (1/3)

Problème : déterminer si un nombre n donné en entrée est premier.

- Algorithme probabiliste polynomial de Rabin-Miller, 1976.
- Algorithme déterministe polynomial de Agrawal-Kayal-Saxena, 2002.

Première tentative : test de Fermat (si n est premier alors $a^{n-1} \equiv 1 \mod n$). Échec à cause des nombres de Carmichael.

Nombres premiers (1/3)

Problème : déterminer si un nombre n donné en entrée est premier.

- Algorithme probabiliste polynomial de Rabin-Miller, 1976.
- Algorithme déterministe polynomial de Agrawal-Kayal-Saxena, 2002.

Soit $n - 1 = 2^k m$ avec k maximal. On dit qu'un nombre a est un témoin (pour savoir si le nombre est composé) si :

- $a^m \not\equiv \pm 1 \mod n$;
- pour tout $1 \leq i \leq k - 1$, $a^{m2^i} \not\equiv -1 \mod n$.

Nombres premiers (2/3)

Lemme

1. *Si n est premier, il n'y a pas de témoin ;*
2. *si n est composé, alors au moins la moitié des entiers $a \in \{1, \ldots, n-1\}$ sont témoins.*

Lemme

1. *Si n est premier, il n'y a pas de témoin ;*
2. *si n est composé, alors au moins la moitié des entiers $a \in \{1, \ldots, n-1\}$ sont témoins.*

Preuve

1. n premier implique $a^{m2^k} \equiv 1 \mod n$ (Fermat). Puisque $(\mathbb{Z}/n\mathbb{Z})$ est intègre, $a^{m2^{k-1}} \equiv \pm 1 \mod n$. Si $\equiv -1$ alors OK, sinon on continue.

Nombres premiers (2/3)

Lemme

1. *Si n est premier, il n'y a pas de témoin ;*
2. *si n est composé, alors au moins la moitié des entiers $a \in \{1, \ldots, n-1\}$ sont témoins.*

Preuve

1. n premier implique $a^{m2^k} \equiv 1 \mod n$ (Fermat). Puisque $(\mathbb{Z}/n\mathbb{Z})$ est intègre, $a^{m2^{k-1}} \equiv \pm 1 \mod n$. Si $\equiv -1$ alors OK, sinon on continue.

2. Les non-témoins forment un sous-groupe de $(\mathbb{Z}/n\mathbb{Z})^*$. Si n est composé, alors ce sous-groupe est strict. □

Nombres premiers (3/3)

Algorithme de Rabin-Miller

1. on choisit $a \in \{1, \ldots, n-1\}$ au hasard ;
2. si a est témoin alors on rejette, sinon on accepte.

Nombres premiers (3/3)

Algorithme de Rabin-Miller

1. on choisit $a \in \{1, \ldots, n-1\}$ au hasard ;
2. si a est témoin alors on rejette, sinon on accepte.

- Si n est premier, alors il n'y a aucun témoin donc l'algorithme acepte.

Nombres premiers (3/3)

Algorithme de Rabin-Miller

1. on choisit $a \in \{1, \ldots, n-1\}$ au hasard ;
2. si a est témoin alors on rejette, sinon on accepte.

- Si n est premier, alors il n'y a aucun témoin donc l'algorithme acepte.
- Si n est composé, alors au moins la moitié des entiers a sont témoins, donc l'algorithme se trompe (i.e. accepte) dans moins de la moitié des cas.

Nombres premiers (3/3)

Algorithme de Rabin-Miller

1. on choisit $a \in \{1, \ldots, n-1\}$ au hasard ;
2. si a est témoin alors on rejette, sinon on accepte.

- Si n est premier, alors il n'y a aucun témoin donc l'algorithme acepte.
- Si n est composé, alors au moins la moitié des entiers a sont témoins, donc l'algorithme se trompe (i.e. accepte) dans moins de la moitié des cas. Comment diminuer la probabilité d'erreur ?

Nombres premiers (3/3)

Algorithme de Rabin-Miller

1. on choisit $a \in \{1, \ldots, n-1\}$ au hasard ;
2. si a est témoin alors on rejette, sinon on accepte.

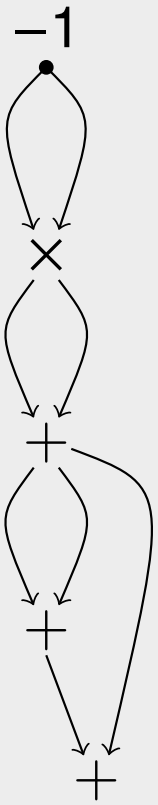
- Si n est premier, alors il n'y a aucun témoin donc l'algorithme acepte.
- Si n est composé, alors au moins la moitié des entiers a sont témoins, donc l'algorithme se trompe (i.e. accepte) dans moins de la moitié des cas. Comment diminuer la probabilité d'erreur ?
- Complexité polynomiale (en la taille de l'entrée, i.e. en $\log n$). Utilisé en pratique car très rapide.

Entier nul (1/4)

Circuits arithmétiques : opérations arithmétiques $+$ et $\times$ à partir de l'entier -1

$\rightarrow$ calcul d'un entier N.

Calcul de $N = 6$

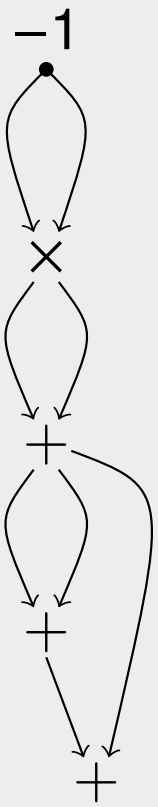


Entier nul (1/4)

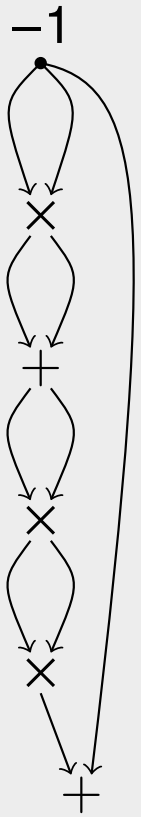
Circuits arithmétiques : opérations arithmétiques $+$ et $\times$ à partir de l'entier -1

$\rightarrow$ calcul d'un entier N.

Calcul de $N = 6$



Calcul de $N = 15$



Entier nul (2/4)

Problème Entier Nul

- Entrée : un circuit arithmétique C calculant un entier N.
- Problème : décider si $N = 0$.

Entier nul (2/4)

Problème Entier Nul

- Entrée : un circuit arithmétique C calculant un entier N.
- Problème : décider si $N = 0$.

Proposer un algorithme fonctionnant en temps polynomial pour ce problème.

Entier nul (2/4)

Problème Entier Nul

- Entrée : un circuit arithmétique C calculant un entier N.
- Problème : décider si $N = 0$.

Proposer un algorithme fonctionnant en temps polynomial pour ce problème.

Difficulté : les nombres calculés peuvent être de taille binaire exponentielle.

Entier nul (3/4)

- Si $|C| = n$, alors $N \leq 2^{2^n}$, donc il a au plus 2^n diviseurs premiers.
- On choisit un nombre premier p entre 2 et 2^{n^2} et on évalue le circuit modulo p.

Entier nul (3/4)

- Si $|C| = n$, alors $N \leq 2^{2^n}$, donc il a au plus 2^n diviseurs premiers.
- On choisit un nombre premier p entre 2 et 2^{n^2} et on évalue le circuit modulo p.

Théorème

Il existe une constante $\alpha > 0$ telle que le nombre $\pi(n)$ de nombres premiers inférieurs à n vérifie $\pi(n) \geq \alpha n / \log n$.

Entier nul (3/4)

- Si $|C| = n$, alors $N \leq 2^{2^n}$, donc il a au plus 2^n diviseurs premiers.
- On choisit un nombre premier p entre 2 et 2^{n^2} et on évalue le circuit modulo p.

Théorème

Il existe une constante $\alpha > 0$ telle que le nombre $\pi(n)$ de nombres premiers inférieurs à n vérifie $\pi(n) \geq \alpha n / \log n$.

Probabilité qu'on tombe sur un diviseur de N :

$$\text{Prob}(p \text{ divise } N) \leq \frac{2^n}{\pi(2^{n^2})} \leq \frac{n^2}{\alpha} 2^{n-n^2} \longrightarrow_{n\to\infty} 0.$$

Entier nul (4/4)

Mais comment trouver un tel nombre premier p ?

Entier nul (4/4)

Mais comment trouver un tel nombre premier p ?
$\rightarrow$ On tire un entier au hasard entre 2 et 2^{n^2} : il est premier avec probabilité $\geq \alpha/n^2 \rightarrow$ on tire $O(n^2)$ entiers.

Entier nul (4/4)

Mais comment trouver un tel nombre premier p ?
$\rightarrow$ On tire un entier au hasard entre 2 et 2^{n^2} : il est premier avec probabilité $\geq \alpha/n^2 \rightarrow$ on tire $O(n^2)$ entiers..

En résumé

1. On répète n^2 fois la procédure suivante :
   - tirer au hasard un entier m compris entre 2 et 2^{n^2} ;
   - évaluer le circuit C modulo m.
2. Si l'une des évaluations est non nulle, alors renvoyer « $N \neq 0$ », sinon renvoyer « $N = 0$ ».

Entier nul (4/4)

Mais comment trouver un tel nombre premier p ?
$\rightarrow$ On tire un entier au hasard entre 2 et 2^{n^2} : il est premier avec probabilité $\geq \alpha/n^2 \rightarrow$ on tire $O(n^2)$ entiers..

En résumé

1. On répète n^2 fois la procédure suivante :
   - tirer au hasard un entier m compris entre 2 et 2^{n^2} ;
   - évaluer le circuit C modulo m.
2. Si l'une des évaluations est non nulle, alors renvoyer « $N \neq 0$ », sinon renvoyer « $N = 0$ ».

- Si $N = 0$ alors l'algorithme donnera la bonne réponse.
- Si $N \neq 0$, l'algorithme donne la bonne réponse avec grande probabilité.

Entier nul (4/4)

Mais comment trouver un tel nombre premier p ?
$\rightarrow$ On tire un entier au hasard entre 2 et 2^{n^2} : il est premier avec probabilité $\geq \alpha/n^2 \rightarrow$ on tire $O(n^2)$ entiers..

En résumé

1. On répète n^2 fois la procédure suivante :
   - tirer au hasard un entier m compris entre 2 et 2^{n^2} ;
   - évaluer le circuit C modulo m.
2. Si l'une des évaluations est non nulle, alors renvoyer « $N \neq 0$ », sinon renvoyer « $N = 0$ ».

- Si $N = 0$ alors l'algorithme donnera la bonne réponse.
- Si $N \neq 0$, l'algorithme donne la bonne réponse avec grande probabilité.

On ne connaît pas d'algorithme déterministe polynomial pour ce problème.

Morale

Le monde se divise en deux catégories :

- les algos probabilistes qui donnent toujours la bonne réponse, mais dont le temps d'exécution dépend du tirage aléatoire ;
- les algos probabilistes qui peuvent se tromper mais avec faible probabilité.

Morale

Le monde se divise en deux catégories :

- les algos probabilistes qui donnent toujours la bonne réponse, mais dont le temps d'exécution dépend du tirage aléatoire ;
- les algos probabilistes qui peuvent se tromper mais avec faible probabilité.

Les algorithmes probabilistes :

- soit accélèrent les algos déterministes connus ;
- soit les simplifient.

Morale

Le monde se divise en deux catégories :

- les algos probabilistes qui donnent toujours la bonne réponse, mais dont le temps d'exécution dépend du tirage aléatoire ;
- les algos probabilistes qui peuvent se tromper mais avec faible probabilité.

Les algorithmes probabilistes :

- soit accélèrent les algos déterministes connus ;
- soit les simplifient.

Peut-on obtenir une accélération superpolynomiale ?

Morale

Le monde se divise en deux catégories :

- les algos probabilistes qui donnent toujours la bonne réponse, mais dont le temps d'exécution dépend du tirage aléatoire ;
- les algos probabilistes qui peuvent se tromper mais avec faible probabilité.

Les algorithmes probabilistes :

- soit accélèrent les algos déterministes connus ;
- soit les simplifient.

Peut-on obtenir une accélération superpolynomiale ?

Mais d'abord : c'est bien beau tout ça mais comment on tire des nombres aléatoires ?

Les nombres aléatoires dans la vraie vie (1/2)

Plusieurs générateurs pseudo-aléatoires populaires

Lehmer (1949, utilisé en C jusqu'à récemment) :

- graine X_0 quelconque (par exemple l'horloge de l'ordinateur) ;
- nombres pseudo-aléatoires suivants :

$$X_{n+1} = (aX_n + b) \mod d, \quad \text{où } d \text{ est } 2^{32};$$

Les nombres aléatoires dans la vraie vie (1/2)

Plusieurs générateurs pseudo-aléatoires populaires

Lehmer (1949, utilisé en C jusqu'à récemment) :

- graine X_0 quelconque (par exemple l'horloge de l'ordinateur) ;
- nombres pseudo-aléatoires suivants :

$$X_{n+1} = (aX_n + b) \mod d, \quad \text{où } d \text{ est } 2^{32};$$

- on peut prendre $d = 2^{32} - 1$ pour éviter des problèmes sur les bits de poids faible ;
- pour avoir une période d, il faut :
  $\mathrm{pgcd}(b, d) = 1$,
  $a - 1$ multiple de p pour tout premier p divisant d,
  et $a - 1$ multiple de 4 si d est multiple de 4.

Les nombres aléatoires dans la vraie vie (2/2)

Plus proche du C maintenant : on favorise les bits de poids fort :
$X'_{n+1} = (aX_n + b)$ avec a très grand, puis

$$X_{n+1} = (X'_{n+1}/c) \mod d \quad \text{où } d = 2^{32} \text{ et } c \text{ constante.}$$

Les nombres aléatoires dans la vraie vie (2/2)

Plus proche du C maintenant : on favorise les bits de poids fort :
$X'_{n+1} = (aX_n + b)$ avec a très grand, puis

$$X_{n+1} = (X'_{n+1}/c) \mod d \quad \text{où } d = 2^{32} \text{ et } c \text{ constante.}$$

Mitchell, Moore (1958, grosso-modo Caml) :

$$X_n = (X_{n-24} + X_{n-55}) \mod d.$$

Période bit de poids faible $2^{55} - 1$,
période globale $2^{e-1}(2^{55} - 1)$ si $d = 2^e$.

Les nombres aléatoires dans la vraie vie (2/2)

Plus proche du C maintenant : on favorise les bits de poids fort :
$X'_{n+1} = (aX_n + b)$ avec a très grand, puis

$$X_{n+1} = (X'_{n+1}/c) \mod d \quad \text{où } d = 2^{32} \text{ et } c \text{ constante.}$$

Mitchell, Moore (1958, grosso-modo Caml) :

$$X_n = (X_{n-24} + X_{n-55}) \mod d.$$

Période bit de poids faible $2^{55} - 1$,
période globale $2^{e-1}(2^{55} - 1)$ si $d = 2^e$.

Sinon : `/dev/random`, quantique...

Les nombres aléatoires dans la vraie vie (2/2)

Plus proche du C maintenant : on favorise les bits de poids fort :
$X'_{n+1} = (aX_n + b)$ avec a très grand, puis

$$X_{n+1} = (X'_{n+1}/c) \mod d \quad \text{où } d = 2^{32} \text{ et } c \text{ constante.}$$

Mitchell, Moore (1958, grosso-modo Caml) :

$$X_n = (X_{n-24} + X_{n-55}) \mod d.$$

Période bit de poids faible $2^{55} - 1$,
période globale $2^{e-1}(2^{55} - 1)$ si $d = 2^e$.

Sinon : `/dev/random`, quantique...

Voyons maintenant ce que ça donne en théorie...

Plan

Définitions

- P : ensemble des langages reconnus par un algorithme déterministe en temps polynomial (en fonction de la taille de l'entrée).

Définitions

- P : ensemble des langages reconnus par un algorithme déterministe en temps polynomial (en fonction de la taille de l'entrée).

- BPP : ensemble des langages reconnus en temps polynomial par un algorithme probabiliste dont la probabilité d'erreur est $\leq 1/3$.

  Exemples : décider si un entier est premier, décider si un circuit arithmétique calcule un entier nul.

Définitions

- P : ensemble des langages reconnus par un algorithme déterministe en temps polynomial (en fonction de la taille de l'entrée).

- BPP : ensemble des langages reconnus en temps polynomial par un algorithme probabiliste dont la probabilité d'erreur est $\leq 1/3$.

  Exemples : décider si un entier est premier, décider si un circuit arithmétique calcule un entier nul.

- Définition formelle de BPP : un langage A est dans BPP s'il existe un langage $B \in \mathrm{P}$ et un polynôme $p(n)$ tel que

$$x \in A \Rightarrow \mathrm{Prob}_{y \in \{0,1\}^{p(n)}}((x,y) \in B) \geq 2/3$$

$$x \notin A \Rightarrow \mathrm{Prob}_{y \in \{0,1\}^{p(n)}}((x,y) \in B) \leq 1/3.$$

Diminution de la probabilité d’erreur

Diminution de la probabilité d'erreur

On répète k fois l'algorithme et on prend la réponse majoritaire.

Diminution de la probabilité d'erreur

On répète k fois l'algorithme et on prend la réponse majoritaire.

Lemme (bornes de Chernoff)

Soient $x_1, \ldots, x_k$ des variables aléatoires prenant les valeurs 1 et 0 avec probabilité p et $1 - p$, et soit $X = \sum_i x_i$ leur somme. Alors

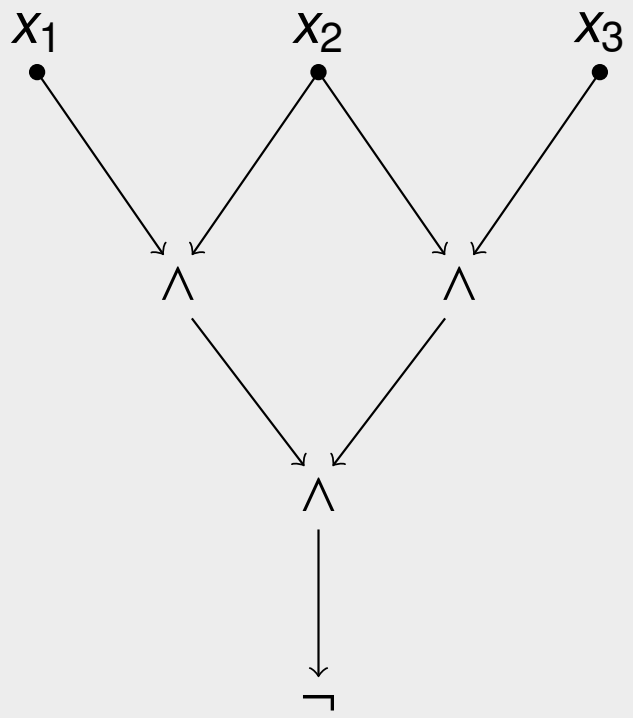
$$\text{Prob}(X \geq (1+\epsilon)pk) \leq e^{-\frac{\epsilon^2}{3}pk}.$$

Après k répétitions, on obtient donc une probabilité d'erreur $\leq e^{-k/36}$.

Après $O(n)$ répétitions, la probabilité d'erreur est $\leq 2^{-n}$.

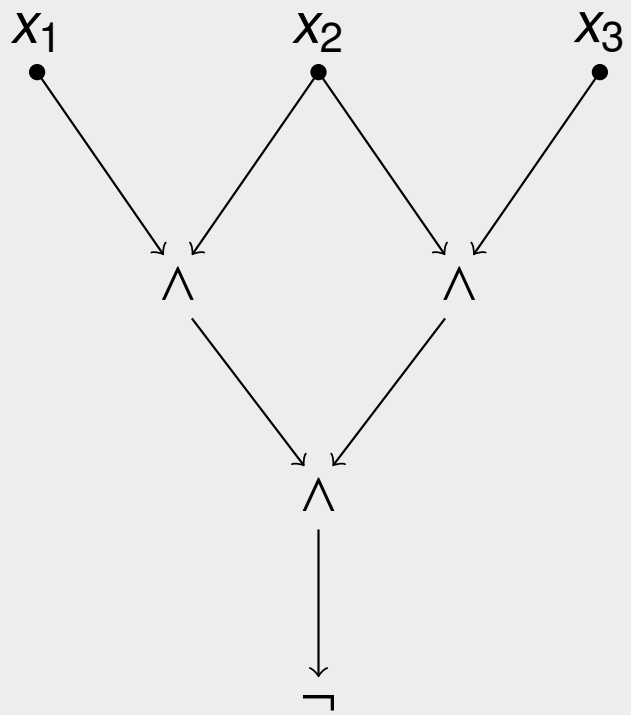
Circuits – classes non-uniformes (1/3)

Circuits booléens : portes $\wedge$, $\vee$ et $\neg$.



Circuits – classes non-uniformes (1/3)

Circuits booléens : portes $\wedge$, $\vee$ et $\neg$.



Pour reconnaître un langage : une famille de circuits (C_n) (un circuit par taille d'entrée).

Circuits – classes non-uniformes (2/3)

- P/poly : langages reconnus par familles de circuits de taille polynomiale (classe non-uniforme).

Circuits – classes non-uniformes (2/3)

- P/poly : langages reconnus par familles de circuits de taille polynomiale (classe non-uniforme). En gros : un algo polynomial pour chaque taille d'entrée.

Circuits – classes non-uniformes (2/3)

- P/poly : langages reconnus par familles de circuits de taille polynomiale (classe non-uniforme). En gros : un algo polynomial pour chaque taille d'entrée.

- Si les circuits peuvent être construits en temps polynomial, alors on obtient la classe P.

Théorème

$\mathrm{BPP} \subset \mathrm{P/poly}$.

Preuve

Soit A dans BPP : il existe un algo polynomial qui se trompe avec probabilité $\leq 2^{-n-1}$. Soit le tableau $T \in \{0,1\}^{n \times p(n)}$ suivant : $T(x,r) = 1$ ssi l'algorithme avec le tirage r se trompe sur l'entrée x.

	x			
r	1	0	$\ldots$	1
	0	0	$\ldots$	0
	$\vdots$	$\vdots$	$\ddots$	$\vdots$
	0	1	$\ldots$	1

Théorème

$\mathrm{BPP} \subset \mathrm{P/poly}$.

Preuve

Soit A dans BPP : il existe un algo polynomial qui se trompe avec probabilité $\leq 2^{-n-1}$. Soit le tableau $T \in \{0,1\}^{n \times p(n)}$ suivant : $T(x,r) = 1$ ssi l'algorithme avec le tirage r se trompe sur l'entrée x.

	x			
r	1	0	...	1
	0	0	...	0
	⋮	⋮	⋱	⋮
	0	1	...	1

La proportion de 1 dans le tableau est $\leq 2^{-n-1}$. Donc il existe une ligne r_0 sans 1 : ce tirage aléatoire r_0 est valide pour toute entrée x. La famille de circuits pour notre problème simule l'algorithme selon ce tirage. □

Plan

Méthode des probabilités conditionnelles (1/3)

- Méthode probabiliste (Erdös) : on montre l'existence d'un élément vérifiant une propriété P en montrant que la probabilité qu'un élément vérifie P est non-nulle.

Méthode des probabilités conditionnelles (1/3)

- Méthode probabiliste (Erdös) : on montre l'existence d'un élément vérifiant une propriété P en montrant que la probabilité qu'un élément vérifie P est non-nulle.

- Méthode des probabilités conditionnelles : à partir d'une telle preuve, en déduire un algorithme déterministe fonctionnant en temps polynomial.

Méthode des probabilités conditionnelles (1/3)

- Méthode probabiliste (Erdös) : on montre l'existence d'un élément vérifiant une propriété P en montrant que la probabilité qu'un élément vérifie P est non-nulle.
- Méthode des probabilités conditionnelles : à partir d'une telle preuve, en déduire un algorithme déterministe fonctionnant en temps polynomial.
- Problème : ça ne marche que pour certains problèmes car on doit être capable de calculer des probabilités. . .

Méthode des probabilités conditionnelles (1/3)

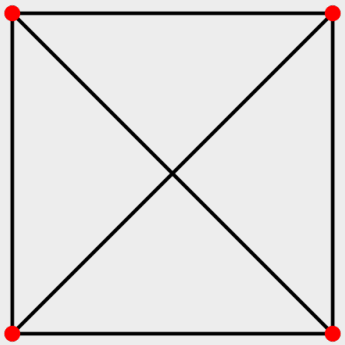
- Méthode probabiliste (Erdös) : on montre l'existence d'un élément vérifiant une propriété P en montrant que la probabilité qu'un élément vérifie P est non-nulle.
- Méthode des probabilités conditionnelles : à partir d'une telle preuve, en déduire un algorithme déterministe fonctionnant en temps polynomial.
- Problème : ça ne marche que pour certains problèmes car on doit être capable de calculer des probabilités. . .

Un exemple et puis c'est bon.

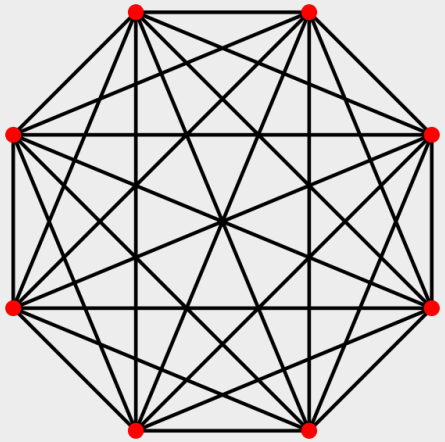
Méthode des probabilités conditionnelles (2/3)

Proposition

Pour tout n, il existe une coloration des arêtes du graphe complet K_n par deux couleurs telle que le nombre de copies monochromatiques de K_4 est $\leq C_n^4/32$.



K_4

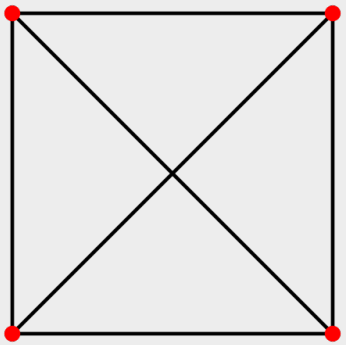


K_8

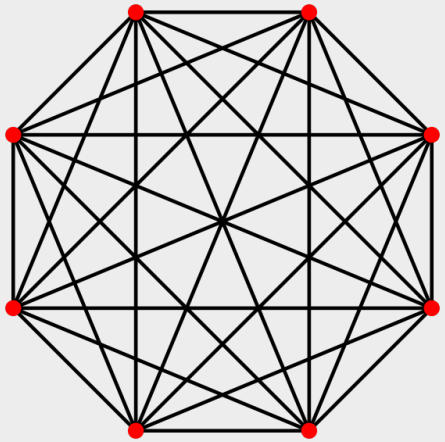
Méthode des probabilités conditionnelles (2/3)

Proposition

Pour tout n, il existe une coloration des arêtes du graphe complet K_n par deux couleurs telle que le nombre de copies monochromatiques de K_4 est $\leq C_n^4/32$.



K_4



K_8

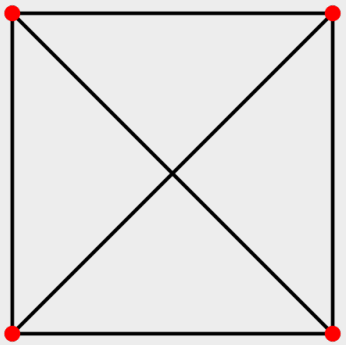
Preuve

Arêtes coloriées avec proba 1/2 entre B et N.

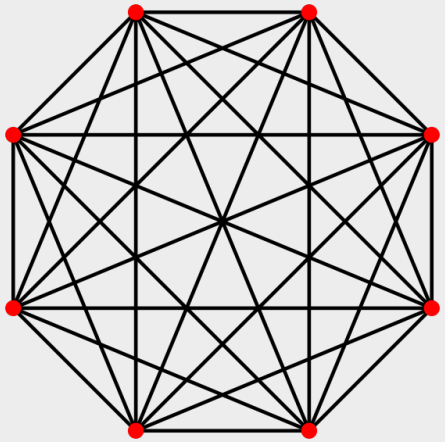
Méthode des probabilités conditionnelles (2/3)

Proposition

Pour tout n, il existe une coloration des arêtes du graphe complet K_n par deux couleurs telle que le nombre de copies monochromatiques de K_4 est $\leq C_n^4/32$.



K_4



K_8

Preuve

Arêtes coloriées avec proba 1/2 entre B et N. Proba qu'une copie de K_4 soit monochromatique : 1/32.

C_n^4 copies de K_4 donc espérance du nombre de copies monochromatiques : $C_n^4/32$. □

Méthode des probabilités conditionnelles (3/3)

Algorithme déterministe pour trouver une coloration des arêtes de K_n possédant moins de $C_n^4/32$ copies monochromatiques de K_4.

Soient $N = C_n^2$ et $e_1, \ldots, e_N$ les arêtes de K_n. On colore les arêtes successivement. Soit X le nombre de copies monochromatiques de K_4 si on termine la coloration aléatoirement.

Méthode des probabilités conditionnelles (3/3)

Algorithme déterministe pour trouver une coloration des arêtes de K_n possédant moins de $C_n^4/32$ copies monochromatiques de K_4.

Soient $N = C_n^2$ et $e_1, \ldots, e_N$ les arêtes de K_n. On colore les arêtes successivement. Soit X le nombre de copies monochromatiques de K_4 si on termine la coloration aléatoirement.

Supposons $e_1, \ldots, e_{i-1}$ colorées ; on veut colorer e_i.

$$E(X|e_1, \ldots, e_{i-1}) = \frac{E(X|e_1, \ldots, e_{i-1}B) + E(X|e_1, \ldots, e_{i-1}N)}{2}$$

donc l'un des deux choix diminue l'espérance. On prend ce choix : ainsi, l'espérance décroît à chaque fois et reste sous $C_n^4/32$.

Méthode des probabilités conditionnelles (3/3)

Algorithme déterministe pour trouver une coloration des arêtes de K_n possédant moins de $C_n^4/32$ copies monochromatiques de K_4.

Soient $N = C_n^2$ et $e_1, \ldots, e_N$ les arêtes de K_n. On colore les arêtes successivement. Soit X le nombre de copies monochromatiques de K_4 si on termine la coloration aléatoirement.

Supposons $e_1, \ldots, e_{i-1}$ colorées ; on veut colorer e_i.

$$E(X|e_1, \ldots, e_{i-1}) = \frac{E(X|e_1, \ldots, e_{i-1}B) + E(X|e_1, \ldots, e_{i-1}N)}{2}$$

donc l'un des deux choix diminue l'espérance. On prend ce choix : ainsi, l'espérance décroît à chaque fois et reste sous $C_n^4/32$.

Il reste à calculer à chaque fois les espérances : on le fait en énumérant toutes les copies de K_4 dans K_n. Temps $O(n^5)$.

Variables aléatoires peu indépendantes

Supposons que l'on a prouvé par la méthode probabiliste l'existence d'un élément en utilisant seulement des variables aléatoires d à d indépendantes. Alors on peut dérandomiser en temps polynomial.

Théorème

Soit d fixé. Pour tout n, on sait construire « efficacement » un espace de taille $O(n^{d/2})$ et n variables aléatoires d à d indépendantes dedans, chacune prenant les valeurs 0 et 1 avec probabilité $1/2$. (Sample space)

Dérandomisation pour l'espace

Théorème

Si un problème de décision a un algorithme probabiliste fonctionnant en espace $s(n)$, alors il a un algorithme déterministe fonctionnant en espace $s(n)^2$.

Plan

Théorème (baratin mou)

- La classe E est l'ensemble des langages reconnus par un algorithme fonctionnant en temps exponentiel $2^{O(n)}$.

Théorème (baratin mou)

- La classe E est l'ensemble des langages reconnus par un algorithme fonctionnant en temps exponentiel $2^{O(n)}$.

- S'il existe une fonction $f : \{0,1\}^* \to \{0,1\}$ dans E difficile à calculer en moyenne (i.e. pas par des circuits de taille polynomiale), alors $\mathrm{BPP} = \mathrm{P}$.

- Idée : utiliser f pour fabriquer un générateur pseudo-aléatoire.

- But : à partir d'un petit nombre de « vrais » bits aléatoires, fabriquer un grand nombre de bits « pseudo-aléatoires ».
- Puisque l'entrée est petite, il peut tourner en temps exponentiel.
- Suite pseudo-aléatoire : aucun algorithme polynomial ne peut la distinguer d'une vraie suite aléatoire.

Definition

Soit $s : \mathbb{N} \to \mathbb{N}$ une fonction. Un s-PRG est une fonction $G : \{0,1\}^* \to \{0,1\}^*$ telle que

- $|G(a)| = s(|a|)$;
- pour tout circuit booléen C à $s(n)$ entrées et de taille $\leq s(n)^3$,

$$\left|\mathrm{Prob}_{a\in\{0,1\}^n}(C(G(a)) = 1) - \mathrm{Prob}_{y\in\{0,1\}^{s(n)}}(C(y) = 1)\right| \leq \frac{1}{10}.$$

Pseudo-aléater implique dérandomiser

Lemme

S'il existe un s-PRG G, alors pour tout fonction t, pour tout algo proba tournant en temps $s(t(n))$, il existe un algo déterministe équivalent tournant en temps $2^{O(t(n))}s(t(n))$.

Pseudo-aléater implique dérandomiser

Lemme

S'il existe un s-PRG G, alors pour tout fonction t, pour tout algo proba tournant en temps $s(t(n))$, il existe un algo déterministe équivalent tournant en temps $2^{O(t(n))}s(t(n))$.

Exemples d'utilisation :

- $s(n) = 2^{\epsilon n}$, $t(n) = O(\log n)$ (dans ce cas, $\mathrm{BPP} = \mathrm{P}$).

Pseudo-aléater implique dérandomiser

Lemme

S'il existe un s-PRG G, alors pour tout fonction t, pour tout algo proba tournant en temps $s(t(n))$, il existe un algo déterministe équivalent tournant en temps $2^{O(t(n))}s(t(n))$.

Exemples d'utilisation :

- $s(n) = 2^{\epsilon n}$, $t(n) = O(\log n)$ (dans ce cas, $\mathrm{BPP} = \mathrm{P}$).
- $s(n) = 2^{n^{\epsilon}}$, $t(n) = O((\log n)^{\alpha})$ (dans ce cas, BPP a des algos déterministes tournant en temps $2^{(\log n)^{O(1)}}$).

Lemme

S'il existe un s-PRG G, alors pour tout fonction t, pour tout algo proba tournant en temps $s(t(n))$, il existe un algo déterministe équivalent tournant en temps $2^{O(t(n))}s(t(n))$.

Preuve

Soit un algo proba A en temps $T = s(t(n))$: tirages aléatoires $r \in \{0,1\}^T \rightarrow$ remplacer r par $G(a)$ où $a \in \{0,1\}^{t(n)}$ uniforme.

Pseudo-aléater implique dérandomiser

Lemme

S'il existe un s-PRG G, alors pour tout fonction t, pour tout algo proba tournant en temps $s(t(n))$, il existe un algo déterministe équivalent tournant en temps $2^{O(t(n))}s(t(n))$.

Preuve

Soit un algo proba A en temps $T = s(t(n))$: tirages aléatoires $r \in \{0,1\}^T \rightarrow$ remplacer r par $G(a)$ où $a \in \{0,1\}^{t(n)}$ uniforme.

Algo déterministe B : on énumère tous les a, on simule A selon le tirage $G(a)$ et on prend la réponse majoritaire. Temps : $2^{t(n)}s(t(n))$.

Lemme

S'il existe un s-PRG G, alors pour tout fonction t, pour tout algo proba tournant en temps $s(t(n))$, il existe un algo déterministe équivalent tournant en temps $2^{O(t(n))}s(t(n))$.

Preuve

Soit un algo proba A en temps $T = s(t(n))$: tirages aléatoires $r \in \{0,1\}^T \to$ remplacer r par $G(a)$ où $a \in \{0,1\}^{t(n)}$ uniforme.

Algo déterministe B : on énumère tous les a, on simule A selon le tirage $G(a)$ et on prend la réponse majoritaire. Temps : $2^{t(n)}s(t(n))$.

Si B se trompe : p.ex. il existe x tq $A(x,r)$ accepte pour $\geq 2/3$ des r mais $A(x,G(a))$ accepte pour moins de la moitié des a.

Alors $|\mathrm{Prob}_r(A(x,r)=1) - \mathrm{Prob}_a(A(x,G(a))=1)| \geq 1/6$, contr. □

Définition

Soit $f : \{0,1\}^n \to \{0,1\}$. *La difficulté (hardness) de* f, *notée* $H(f)$, *est le plus grand entier* h *tel que tout circuit booléen de taille* $\leq h$ *à* n *entrées satisfasse*

$$\text{Prob}_{x \in \{0,1\}^n}(C(x) = f(x)) < \frac{1}{2} + \frac{1}{h}.$$

Définition

Soit $f : \{0,1\}^n \to \{0,1\}$. La difficulté (hardness) de f, notée $H(f)$, est le plus grand entier h tel que tout circuit booléen de taille $\leq h$ à n entrées satisfasse

$$\mathrm{Prob}_{x\in\{0,1\}^n}(C(x) = f(x)) < \frac{1}{2} + \frac{1}{h}.$$

Exemple : si $f \in \mathrm{P/poly}$ alors f a des circuits de taille polynomiale donc $H(f)$ est bornée par un polynôme.

Définition

Soit $f : \{0,1\}^n \to \{0,1\}$. La difficulté (hardness) de f, notée $H(f)$, est le plus grand entier h tel que tout circuit booléen de taille $\leq h$ à n entrées satisfasse

$$\mathrm{Prob}_{x\in\{0,1\}^n}(C(x) = f(x)) < \frac{1}{2} + \frac{1}{h}.$$

Exemple : si $f \in \mathrm{P/poly}$ alors f a des circuits de taille polynomiale donc $H(f)$ est bornée par un polynôme.

Si la cryptographie est sûre, alors il existe des fonctions de NP ayant une difficulté super-polynomiale.

Et pour quelques bits de plus

Lemme

Supposons qu'il existe $f : \{0,1\}^* \to \{0,1\}$ *de difficulté* $H(f) \geq n^4$.
Alors il existe un s-PRG G pour $s(n) = n + 1$.

Lemme

Supposons qu'il existe $f : \{0,1\}^* \to \{0,1\}$ *de difficulté* $H(f) \geq n^4$. *Alors il existe un s-PRG G pour* $s(n) = n + 1$.

Preuve

Soit $G(a) = af(a)$. Supposons que G n'est pas un s-PRG : alors il existe un circuit C de taille $\leq (n+1)^3$ tel que

$$\mathrm{Prob}_{a \in \{0,1\}^n}(C(af(a)) = 1) - \mathrm{Prob}_{y \in \{0,1\}^{n+1}}(C(y)) > \frac{1}{10}.$$

Lemme

Supposons qu'il existe $f : \{0,1\}^* \to \{0,1\}$ *de difficulté* $H(f) \geq n^4$. *Alors il existe un s-PRG G pour* $s(n) = n + 1$.

Preuve

Soit $G(a) = af(a)$. Supposons que G n'est pas un s-PRG : alors il existe un circuit C de taille $\leq (n+1)^3$ tel que

$$\mathrm{Prob}_{a \in \{0,1\}^n}(C(af(a)) = 1) - \mathrm{Prob}_{y \in \{0,1\}^{n+1}}(C(y)) > \frac{1}{10}.$$

On a envie de dire que l'une des deux fonctions suivantes « approxime » f :

$$f_1(a) = \begin{cases} 0 & \text{si } C(a0) = 1 \\ 1 & \text{sinon} \end{cases} \qquad f_2(a) = \begin{cases} 1 & \text{si } C(a1) = 1 \\ 0 & \text{sinon} \end{cases}$$

Suite de la preuve

Mais on ne sait pas choisir entre f_1 et $f_2 \rightarrow$ on prend la moyenne !

f' : on tire $r \in \{1, 2\}$ au hasard et on renvoie $f_r(a)$.

Suite de la preuve

Mais on ne sait pas choisir entre f_1 et $f_2 \rightarrow$ on prend la moyenne !
f' : on tire $r \in \{1, 2\}$ au hasard et on renvoie $f_r(a)$.

Ainsi,

$$\mathrm{Prob}_{ar}(f'(a) = f(a)) = \frac{1}{2}\Big(\mathrm{Prob}_a(f_1(a) = f(a)) + \mathrm{Prob}_a(f_2(a) = f(a))\Big).$$

Donc si on arrive à montrer que $\mathrm{Prob}_{ar}(f'(a) = f(a)) > 1/2 + \epsilon$ alors ce sera aussi le cas de f_1 ou f_2 et donc f serait approximable par un circuit de taille $< n^4$, contradiction.

Suite de la preuve

Mais on ne sait pas choisir entre f_1 et $f_2 \rightarrow$ on prend la moyenne !
f' : on tire $r \in \{1, 2\}$ au hasard et on renvoie $f_r(a)$.

Ainsi,

$$\mathrm{Prob}_{ar}(f'(a) = f(a)) = \frac{1}{2}\Big(\mathrm{Prob}_a(f_1(a) = f(a)) + \mathrm{Prob}_a(f_2(a) = f(a))\Big).$$

Donc si on arrive à montrer que $\mathrm{Prob}_{ar}(f'(a) = f(a)) > 1/2 + \epsilon$ alors ce sera aussi le cas de f_1 ou f_2 et donc f serait approximable par un circuit de taille $< n^4$, contradiction.

Il suffit donc de montrer $\mathrm{Prob}_{ar}(f'(a) = f(a)) > 1/2 + \epsilon$.

Suite de la preuve

Cas où $f'(a) = f(a)$:

- $C(ar) = 1$ et $r = f(a)$, ou
- $C(ar) = 0$ et $r = \overline{f(a)}$

Ces cas arrivent plus fréquemment que le contraire car l'inégalité

$$\mathrm{Prob}_{a\in\{0,1\}^n}(C(af(a)) = 1) - \mathrm{Prob}_{y\in\{0,1\}^{n+1}}(C(y)) > \frac{1}{10}$$

implique

$$\mathrm{Prob}_{a\in\{0,1\}^n}(C(af(a)) = 1) - \mathrm{Prob}_{y\in\{0,1\}^{n+1}}(C(a\overline{f(a)}) = 1) > \frac{1}{5}.$$

Ainsi, $\mathrm{Prob}_{ar}(f'(a) = f(a)) > 1/2 + 1/10$ et f n'était pas difficile en moyenne : contradiction. □

PRG exponentiel

Théorème (Nisan, Wigderson 1994)

Soit $s : \mathbb{N} \to \mathbb{N}$ *une fonction. S'il existe* $f : \{0,1\}^* \to \{0,1\}$ *de difficulté* $H(f) \geq s(n)$*, alors il existe un* s^ϵ*-PRG.*

Théorème (Nisan, Wigderson 1994)

Soit $s : \mathbb{N} \to \mathbb{N}$ *une fonction. S'il existe* $f : \{0,1\}^* \to \{0,1\}$ *de difficulté* $H(f) \geq s(n)$*, alors il existe un* s^ϵ*-PRG.*
En particulier :

- *s'il existe* $f \in \mathrm{E}$ *telle que* $H(f) \geq 2^{\epsilon n}$*, alors* $\mathrm{BPP} = \mathrm{P}$ *;*
- *s'il existe* $f \in \mathrm{E}$ *telle que* $H(f) \geq 2^{n^\epsilon}$*, alors* BPP *a des algos déterministes tournant en temps* $2^{(\log n)^{O(1)}}$*.*

Théorème (Nisan, Wigderson 1994)

Soit $s : \mathbb{N} \to \mathbb{N}$ *une fonction. S'il existe* $f : \{0,1\}^* \to \{0,1\}$ *de difficulté* $H(f) \geq s(n)$, *alors il existe un* s^ϵ*-PRG.*
En particulier :

- *s'il existe* $f \in \mathrm{E}$ *telle que* $H(f) \geq 2^{\epsilon n}$, *alors* $\mathrm{BPP} = \mathrm{P}$ *;*
- *s'il existe* $f \in \mathrm{E}$ *telle que* $H(f) \geq 2^{n^\epsilon}$, *alors* BPP *a des algos déterministes tournant en temps* $2^{(\log n)^{O(1)}}$.

Preuve

Construction d'un s^ϵ-PRG à partir de f.

NW-generator sur l'entrée $a \in \{0,1\}^n$: si $\mathcal{I} = \{I_1, \ldots, I_m\}$ sont des sous-ensembles de $\{1, \ldots, n\}$ alors

$$\mathrm{NW}_{\mathcal{I}}(a) = f(a_{|I_1}) f(a_{|I_2}) \ldots f(a_{|I_m}).$$

Suite de la preuve

Définition

Si $n > \ell > d$ *alors* $\mathcal{I} = \{I_1, \ldots, I_m\}$ *(sous-ensembles de* $\{1, \ldots, n\}$*)*
est un (n, ℓ, d)*-design si* $\forall i, |I_i| = \ell$ *et* $\forall j \neq k, |I_j \cap I_k| \leq d$.

Suite de la preuve

Définition

Si $n > \ell > d$ alors $\mathcal{I} = \{I_1, \ldots, I_m\}$ (sous-ensembles de $\{1, \ldots, n\}$) est un (n, ℓ, d)-design si $\forall i, |I_i| = \ell$ et $\forall j \neq k, |I_j \cap I_k| \leq d$.

Lemme

Si $\mathcal{I}$ est un (n, ℓ, d)-design avec $|\mathcal{I}| \geq 2^{d/10}$ et $f : \{0,1\}^* \to \{0,1\}$ avec $H(f) > 2^{3d}$, alors $\mathrm{NW}_{\mathcal{I}}$ est un 2^d-PRG.

Suite de la preuve

Définition

Si $n > \ell > d$ alors $\mathcal{I} = \{I_1, \ldots, I_m\}$ (sous-ensembles de $\{1, \ldots, n\}$) est un (n, ℓ, d)-design si $\forall i, |I_i| = \ell$ et $\forall j \neq k, |I_j \cap I_k| \leq d$.

Lemme

Si $\mathcal{I}$ est un (n, ℓ, d)-design avec $|\mathcal{I}| \geq 2^{d/10}$ et $f : \{0,1\}^* \to \{0,1\}$ avec $H(f) > 2^{3d}$, alors $\mathrm{NW}_{\mathcal{I}}$ est un 2^d-PRG.

Lemme

On sait construire des (n, ℓ, d)-designs efficacement.

$\to$ on obtient un PRG qui fabrique un nombre exponentiel de bits pseudo-aléatoires. □

Et après ?

En utilisant un résultat de Yao, on sait construire une fonction difficile en moyenne à partir d'une fonction difficile dans le pire cas.

Et après ?

En utilisant un résultat de Yao, on sait construire une fonction difficile en moyenne à partir d'une fonction difficile dans le pire cas. Et on sait même dérandomiser le résultat de Yao. On obtient alors :

Théorème (Impagliazzo, Wigderson 1997)

Si E contient un langage nécessitant (presque partout) des circuits de taille $2^{\Omega(n)}$*, alors* $\mathrm{P} = \mathrm{BPP}$.

Dans l'autre direction

Il existe aussi un résultat montrant que dérandomiser revient à montrer des bornes inférieures non-uniformes.

Théorème (Impagliazzo, Kabanets 2003)

Si Entier Nul $\in \mathrm{P}$ *alors l'une des conditions suivantes est vraie :*

- $\mathrm{NEXP} \not\subset \mathrm{P/poly}$ *;*
- *le permanent n'est pas calculable par circuits arithmétiques de taille polynomiale.*

Remarque : l'hypothèse est impliquée par $\mathrm{BPP} = \mathrm{P}$.

Conclusion

Conclusion

- Les algorithmes probabilistes aident vraiment.

Conclusion

- Les algorithmes probabilistes aident vraiment.
- Les algorithmes probabilistes aident-ils vraiment ?

Conclusion

- Les algorithmes probabilistes aident vraiment.
- Les algorithmes probabilistes aident-ils vraiment ?
- On croit aux bornes inférieures non-uniformes donc on croit à la dérandomisation.

Conclusion

- Les algorithmes probabilistes aident vraiment.
- Les algorithmes probabilistes aident-ils vraiment ?
- On croit aux bornes inférieures non-uniformes donc on croit à la dérandomisation.
- Liens forts circuits / algorithmes probabilistes. But : montrer qu'on peut dérandomiser BPP, ou montrer des bornes inférieures non-uniformes.

Conclusion

- Les algorithmes probabilistes aident vraiment.
- Les algorithmes probabilistes aident-ils vraiment ?
- On croit aux bornes inférieures non-uniformes donc on croit à la dérandomisation.
- Liens forts circuits / algorithmes probabilistes. But : montrer qu'on peut dérandomiser BPP, ou montrer des bornes inférieures non-uniformes.

Bonne nuit !