Die Themen

1. Welche Eigenschaften sollte ein Pseudo-Random Generator haben?
   - Er sollte von wirklichen Zufallsgeneratoren nicht unterscheidbar sein?!
   - Eine viel zu starke Forderung: Stattdessen
     - ★ sollte ein „vernünftiger Test“ den Pseudo-Random Generator nicht vom wirklichen Zufallsgenerator unterscheiden können.
     - ★ Was sind vernünftige Tests?
2. Kann Randomisierung eine super-polynomielle Leistungssteigerung gegenüber deterministischen Algorithmen bringen?
   - Oder können deterministische Algorithmen randomisierte Algorithmen mit Hilfe von Pseudo-Random Generatoren nachmachen?

Zufallsbits

Wie kommen wir an Zufallsbits?

- Das Ausnutzen physikalische Effekte wie
  - das thermische Rauschen eines Widerstands oder
  - radioaktive Zerfallsvorgänge

  liefert hervorragende Generatoren. Aber:
  - Auf Schnelligkeit getrimmte Messgeräte müssen gebaut werden.
  - Die Reproduzierbarkeit eines Zufallsexperiments ist nicht gewährleistet.
- In der Praxis verwendet man Pseudo-Random Generatoren:
  - Aus einer „zufälligen Saat“, wie etwa der Systemzeit, wird eine zufällig erscheinende Bitfolge gebaut.
  - Der Konstruktionsprozess erfolgt durch einen deterministischen Algorithmus.

Welche Anforderungen muss ein guter Pseudo-Random Generator erfüllen? Gibt es gute Pseudo-Random Generatoren?

Generatoren

Sei p ein echt monoton wachsendes Polynom mit $p(n) > n$.

- Ein **Generator**

$$G : \{0,1\}^* \to \{0,1\}^*$$

  mit Streckung p produziert auf binären Eingaben der Länge n binäre Ausgaben der Länge $p(n)$.
- G heißt **effizient**, wenn es eine in Polynomialzeit rechnende deterministische Turingmaschine gibt, die den Generator implementiert.

Ein Generator soll aus wenigen Zufallsbits viele machen.

Statistische Tests

Ein statistischer Test $\mathcal{T}$ ist ein randomisierter Algorithmus, der eine Eingabe akzeptiert oder verwirft.

- Wann besteht ein Generator (mit Streckung $p(n)$) den Test $\mathcal{T}$?

  Wir setzen

$$\begin{aligned} g_n &= \text{prob}[\ T \text{ akzeptiert } G(x) \mid |x| = n\ ] \text{ und} \\ r_n &= \text{prob}[\ T \text{ akzeptiert } y \mid |y| = p(n)\ ]. \end{aligned}$$

  Für die Bestimmung von g_n verwende die Gleichverteilung auf den Paaren (x, u) mit $x \in \{0,1\}^n$ und u als Folge der von T auf Eingabe $G(x)$ angeforderten Zufallsbits.

- G besteht den Test T, wenn es zu jedem $k \in \mathbb{N}$ eine Schranke N_k gibt, so dass

$$\forall n \geq N_k \quad |g_n - r_n| \leq n^{-k}.$$

Wie schwierig dürfen Tests sein?

Gibt es Generatoren G, die alle statistischen Tests bestehen?

- G strecke Eingaben der Länge n auf Ausgaben der Länge $p(n) > n$.
- Betrachte den Test

$$T(y) = \begin{cases} 1 & G \text{ produziert } y \text{ als Ausgabe,} \\ 0 & \text{sonst.} \end{cases}$$

- Es ist
  - $\text{prob}[\, G(x) \in T \mid |x| = n \,] = 1$,
  - während ein wirklicher Zufallsgenerator höchstens die Wahrscheinlichkeit $\frac{1}{2}$ erzielt.

Es gibt keinen Generator, der alle statistischen Tests besteht!

Pseudo-Random Generatoren

Unser Hauptinteresse sollte dem Einsatz von Generatoren in effizienten randomisierten Algorithmen A gelten.

- A definiert ebenfalls einen statistischen Test, aber diesmal ist der Test effizient!

Ein (effizienter) Generator G mit Streckung $p(n) > n$ heißt ein (effizienter) Pseudo-Random Generator, wenn G **jeden effizienten** statistischen Test besteht.

Jede polynomielle Streckung ist erreichbar:
Ersetze G durch $G(G(x))$, bzw. $G(G(G(x)))$, bzw $G^{\text{poly}}(x)$.

Lineare Kongruenzen

Ist der Generator der linearen Kongruenzen ein Pseudo-Random Generator?

- Für eine „Saat“ s, einem Modulus m, einem Koeffizienten a und einem Offset b wird die Folge

$$x_0 = s, \quad x_{k+1} \equiv (a \cdot x_k + b) \mod m.$$

berechnet.

- Der Generator wird häufig benutzt, ist aber kein Pseudo-Random Generator!
  - Im 2-dimensionalen: Alle Paare (x_i, x_{i+1}) liegen auf höchstens $(2! \cdot m)^{1/2}$ parallelen Geraden.
  - Im n-dimensionalen: Alle Vektoren $(x_i, \ldots, x_{i+n-1})$ liegen auf höchstens $(n! \cdot m)^{1/n}$ parallelen Hyperebenen.

    $(n! \cdot m)^{1/n} \leq (n^n \cdot m)^{1/n} = n \cdot m^{1/n}$.

Gibt es effiziente Pseudo-Random Generatoren?

Wenn $\mathcal{NP} \subseteq \mathcal{BPP}$ oder $\mathcal{NP} = \mathcal{P}$, dann gibt es keine effizienten Pseudo-Random Generatoren.

- Es gelte $\mathcal{NP} \subseteq \mathcal{BPP}$ oder $\mathcal{NP} = \mathcal{P}$ und G sei ein effizienter Pseudo-Random Generator.
- Wir bauen einen effizienten statistischen Test:
  - Die Sprache $L = \{G(x) \mid x \in \{0,1\}^*\}$ liegt in $\mathcal{NP}$:
    - ⋆ Für Eingabe y rate x, berechne $G(x)$ und akzptiere, falls $y = G(x)$.
  - Nach Annahme liegt aber L auch in $\mathcal{BPP}$ bzw. $\mathcal{P}$.
    - ⋆ Unser statistischer Test akzeptiert y genau dann, wenn $y \in L$, wenn also $y = G(x)$.

Der Nachweis, dass es Pseudo-Random Generatoren gibt, sollte uns schwerfallen, denn dann zeigen wir auch $\mathcal{P} \neq \mathcal{NP}$.

One-way Funktionen

Es gibt einen starken Zusammenhang zwischen Pseudo-Random Generatoren und One-way Funktionen.

Eine Funktion $f : \{0,1\}^* \to \{0,1\}^*$ heißt eine **One-Way Funktion**, wenn

f in polynomieller Zeit durch eine deterministische Turingmaschine berechenbar ist und wenn

$$\text{prob} \quad [\quad M \text{ berechnet auf Eingabe } f(x) \text{ ein } z \text{ mit } f(z) = f(x) \mid x \in \{0,1\}^n \;] \; < \frac{1}{p(n)}$$

für **jedes** Polynom p, für **jede** probabilistische Turingmaschine M, die in Zeit $O(p)$ rechnet, und für **alle** hinreichend großen n gilt.

One-way Funktionen lassen sich nicht effizient von randomisierten Algorithmen invertieren.

One-Way Funktionen und Pseudo-Random Generatoren

Die folgenden Aussagen sind äquivalent:

(a) Es gibt einen effizienten Pseudo-Random Generator G mit Streckung $p(n) > n$.

(b) Es gibt one-way Funktionen.

- (a) $\Rightarrow$ (b): Wenn wir G invertieren könnten, dann könnten wir einen effizienten Test bauen, den G nicht besteht:

  Der Test akzeptiert y genau dann, wenn es x mit $y = G(x)$ gibt.
- (b) $\Rightarrow$ (a): Diese Implikation ist wesentlich komplizierter.

Wie baut man Pseudo-Random Generatoren aus One-way Funktionen?

Der diskrete Logarithmus

Für eine **Primzahl** p, ein **erzeugendes Element** g modulo p und eine natürliche Zahl i berechne die Potenz

$$g^i \bmod p.$$

- Das Umkehrproblem ist das diskrete Logarithmus Problem:
  Für Eingabe p, g und $y \mod p$ berechne den Logarithmus von y, also die Potenz i mit $y \equiv g^i \mod p$.
- Der Blum-Micali (BM) Generator:
  - wähle eine Primzahl p und eine erzeugende Restklasse $g \mod p$.
  - Für eine Saat s_0 berechne die Iteration $s_{i+1} = g^{s_i} \bmod p$ und gib die Bitfolge $(b_1, \ldots, b_m)$ aus, wobei

$$b_i = \begin{cases} 1 & \text{wenn } s_i < p/2, \\ 0 & \text{sonst.} \end{cases}$$

Der BM-Generator basiert also auf dem diskreten Logarithmus als One-Way Funktion.

Das RSA-Problem

N ist ein Produkt von zwei unbekannten Primzahlen.

Für Eingabe N, e und x,

wobei e und $\phi(N)$, die Anzahl der primen Restklassen modulo N, teilerfremd sind,

berechne $y \equiv x^e \mod N$.

- Das Umkehrproblem ist das RSA-Problem:

  Für Eingabe N, e und y bestimme x mit $y \equiv x^e \mod N$.
- Der RSA Generator:

  Für eine Saat s_0 berechne $s_{i+1} = s_i^e \mod N$ und gib die Bitfolge $(s_1 \mod 2, \ldots, s_m \mod 2)$ aus.

  Der RSA-Generator beruht auf dem RSA-Problem als One-Way Funktion.

Die diskrete Quadratwurzel-Berechnung

Für Eingaben x und m mit $x < m$ berechne $x^2 \mod m$.

- Das Umkehrproblem ist die diskrete Quadratwurzel-Berechnung:
  - Für Eingabe m und y berechne x mit $y \equiv x^2 \mod m$.
- Der Blum-Blum-Shub Generator:

  Für eine Saat s_0 berechne $s_{i+1} = s_i^2 \mod m$,

  wobei $m = p \cdot q$ mit Primzahlen $p \equiv q \equiv 3 \mod 4$ gelte.

  Der Generator gibt die Bitfolge $(s_1 \mod 2, \ldots, s_m \mod 2)$ aus.

Der BBS-Generator beruht auf der diskreten Quadratwurzelberechnung als One-Way Funktion.

Fazit

Der diskrete Logarithmus, das RSA-Problem und die diskrete Quadratwurzelberechnung sind „wahrscheinlich“ One-Way Funktionen.

+ Der Blum-Micali Generator, der RSA-Generator und der Blum-Blum-Shub Generator sind „wahrscheinlich“ Pseudo-Random Generatoren.
- Ihre große Schwäche ist die recht langsame Berechnung.

- In praktischen Anwendungen ist der Generator der linearen Kongruenzen unverwüstlich, aber nicht sicher.
- Der Mersenne Twister ist schnell und vermutlich sicherer.

Derandomisierung

Sei M eine probabilistische Turingmaschine mit Laufzeit n^r.

- M wird höchstens n^r Zufallsbits anfordern.
- G sei ein Pseudo-Random Generator mit Streckung $p(n) = n^k$.
  - Für jeden 0-1 String x der Länge $n^{r/k}$ benutze $G(x)$ als Zufallsquelle und simuliere M deterministisch.
    - ⋆ $G(x)$ ist ein 0-1 String der Länge $(n^{r/k})^k = n^r$.
  - Akzeptiere genau dann, wenn die meisten Simulationen akzeptieren.

Unsere deterministische Simulation hat die Laufzeit

$$O(n^r \cdot 2^{n^{r/k}}).$$

Derandomisierung II

Ist ein Pseudo-Random Generator ausreichend?

- Wir fixieren die Eingabe w von M und erhalten einen effizienten statistischen Test mit „Orakelstring“ w:
  - Simuliere M auf Eingabe y und w.
  - Akzeptiere genau dann, wenn M „seine“ Eingabe w mit y als Zufallssequenz akzeptiert.
- G muss schwierigere statistische Tests, nämlich nicht-uniforme Tests bestehen:
  - Der Test entspricht einem randomisierten Algorithmus und einer zusätzlichen Eingabe –nämlich w– für den Algorithmus.

Wenn es Pseudo-Random Generatoren gibt, die auch alle effizienten, nicht-uniformen Tests bestehen, dann ist

$$\mathcal{BPP} \subseteq \bigcap_{\varepsilon>0} \mathrm{DTIME}(2^{n^\varepsilon}).$$

$\mathcal{P}$ versus $\mathcal{BPP}$

Es gibt starke Indizien für die Gleichheit

$$\mathcal{P} = \mathcal{BPP}.$$

- Sei $\mathcal{E}$ die Komplexitätsklasse aller Sprachen, die durch deterministische Turingmaschinen in Zeit $2^{O(n)}$ erkannt werden können.
- Wenn es eine Funktion $f \in \mathcal{E}$ gibt, so dass f nur durch Schaltkreise der Größe $2^{\Omega(n)}$ berechenbar ist, dann folgt $\mathcal{P} = \mathcal{BPP}$.
- Dies ändert nichts daran, dass
  - randomisierte Algorithmen häufig einfacher und (polynomiell) schneller als deterministische Algorithmen sind
  - und beweisbar besser sind, wenn die Eingabe nicht vollständig bekannt ist.