Probabilistische Algorithmen

Michal Švancar Gerardo Balderas

Hochschule Zittau/Görlitz

21. Dezember 2014

Inhaltsverzeichnis

Einleitung

Wir sind auf der Suche nach besseren und effizienteren Algorithmen.
Man kann bei manchen Algorithmen einen Zufallswert einfügen und damit diese verbessern.
Man nennt solche Algorithmen:

Probabilistische Algorithmen

Pseudozufallszahlen:

Zufallszahlen, die in Wirklichkeit nicht zufällig sind

Pseudozufallszahlen

- sind deterministisch
- werden durch einen Generator erzeugt
  - man nennt diese Generatoren RNG (Random Number Generator)
- sind fast in allen Programmiersprachen integriert
- sind normalerweise mit einer Funktion *random* verfügbar

Kongruenzmethode

- 1948 von Derrick Henry Lehmer entwickelt
- RNGs, die mit dieser Methode entwickelt sind, heißen Kongruenzgeneratoren
- hat den Vorteil, einen eigenen Generator herzustellen
- hat eine rekursive Vorschrift

Der Kongruenzgenerator

LCG (Linear Congruential Generator)

$$z_n = (a \cdot z_{n-1} + b) \mod c$$

z_0 wird als seed(Saat) bezeichnet
a ist der 'multiplier'
b ist der 'increment'
c ist modulo

Erzeugte Zahlen scheinen zufällig gewählt zu sein, sind es aber nicht. Man kann mathematisch den Algorithmus 'knacken'.

Periodenlänge

Um eine Periodenlänge von maximal c zu haben ist es notwendig:

- dass b und c teilerfremd sind
- dass $a - 1$ durch alle Prim-Faktoren von c teilbar ist
- wenn $a - 1$ durch 4 teilbar ist, dann muss m auch durch 4 teilbar sein

Da die Periodenlänge von c abhängt, wird immer empfohlen, einen Wert von $2^{31} - 1$ oder 2.147.483.647 zu nehmen.

Bei wiederholter Ausführung des RNG mit einem festgestellten *Seed*, werden immer die gleichen Werte an den gleichen Stellen ausgegeben, deswegen ist er nicht für Zufallsexperimenten geeignet.
Dafür wird ein zufälliger *Seed* genommen: **ZEIT**.
Durch die Eingabe der aktuellen Zeit des Rechners wird der Generator immer etwas anderes erzeugen.

Bibliothek	c	a	b
C/C++	2^{31}	1103515245	12345
Visual Basic	2^{32}	1140671485	12820163
Java	2^{48}	25214903917	11

Andere RNG Methoden

- Inverser Kongruenzgenerator
- Mersenne-Twister (häufigster RNG)
- Multiply-with-Carry (extrem lange Periode [bis $2^{2000000}$])
- Xorshift (kleine Mengen von Zufallszahlen, brauchen wenig Speicherplatz)

Numerische probabilistische Algorithmen

u.A Zufallsexperimente, Tests, *Nichtdeterministische Simulationen*

Klassisches Beispiel: Näherung für die Zahl π

- 'Zufallsregen' (zufällige Punkte erzeugen)
- von erzeugten Punkten zählen, wie viele Punkte im Kreis und außerhalb liegen

$$Fläche = \frac{Anz_{imKreis}}{Anz_{total}}$$
$$\text{Fläche des Quadrats} = x^2$$
$$\text{Fläche des Kreises} = \pi r^2$$
$$\frac{Anz_{imKreis}}{Anz_{total}} = \frac{\pi}{4}$$
$$\frac{Anz_{imKreis}}{Anz_{total}} \cdot 4 \approx \pi$$

beim ersten Quadrant gilt:

Für Punkte innerhalb des Kreises $x^2 + y^2 \leq 1$, für reele x und y mit $0 < x, y \leq 1$

Um die Fläche einer Funktion zu berechnen, braucht man ein Integral.
Zufallszahlen können den Wert ohne Integrieren bekommen.
Man macht einen zufälligen Punkt zwischen die Werte die man berechnen möchte.
Wenn der Punkt unter der Funktion liegt, ist er ein treffender Punkt.

$$\frac{\text{Fläche unter die Kurve}}{\text{Fläche von der Rechtseck}} \approx \frac{\text{Treffenderpunkte}}{\text{Gesamtepunkte}}$$

Las-Vegas-Algorithmen

- Las-Vegas-Algorithmen sind ein Art von probabilistischen Algorithmen, ähnlich wie Monte-Carlo Algorithmen

Las-Vegas-Algorithmen

- Las-Vegas-Algorithmen sind ein Art von probabilistischen Algorithmen, ähnlich wie Monte-Carlo Algorithmen
- wurden von László Babai in 1979 im Kontext mit Graphenisomorphie entwickelt

Las-Vegas-Algorithmen

- Las-Vegas-Algorithmen sind ein Art von probabilistischen Algorithmen, ähnlich wie Monte-Carlo Algorithmen
- wurden von László Babai in 1979 im Kontext mit Graphenisomorphie entwickelt
- sie verändern nicht die Richtigkeit des Ergebnisses, sondern die Menge der benötigten Berechungsressourcen
- Las Vegas ist eine bekannte Stadt für Gambling

Las-Vegas-Algorithmen

- Las-Vegas Algorithmen liefern **niemals** ein falsches Ergebnis

Las-Vegas-Algorithmen

- Las-Vegas Algorithmen liefern **niemals** ein falsches Ergebnis
- Sie können zu einer Sackgasse führen

Las-Vegas-Algorithmen

- Las-Vegas Algorithmen liefern **niemals** ein falsches Ergebnis
- Sie können zu einer Sackgasse führen
- Bei der wiederholten Ausführung des Algorithmus kann er mit neuen Zufallszahlen doch ein richtiges Ergebnis liefern

Las-Vegas-Algorithmen

- Las-Vegas Algorithmen liefern **niemals** ein falsches Ergebnis
- Sie können zu einer Sackgasse führen
- Bei der wiederholten Ausführung des Algorithmus kann er mit neuen Zufallszahlen doch ein richtiges Ergebnis liefern
  - Jede wiederholte Ausführung erhöht die Wahrscheinlichkeit eines korrekten Ergebnisses

Las-Vegas-Algorithmen

- Las-Vegas Algorithmen liefern **niemals** ein falsches Ergebnis
- Sie können zu einer Sackgasse führen
- Bei der wiederholten Ausführung des Algorithmus kann er mit neuen Zufallszahlen doch ein richtiges Ergebnis liefern
  - Jede wiederholte Ausführung erhöht die Wahrscheinlichkeit eines korrekten Ergebnisses
  - Nach einer bestimmten Anzahl von Wiederholungen ist es sehr unwahrscheinlich, kein korrektes Ergebnis zu erhalten

Las-Vegas-Aufwandsanalyse

- Für die Aufwandsanalyse wird als Beispiel eine abstrakte Methode $LV(x)$ zur Lösung des Problems x genommen

Las-Vegas-Aufwandsanalyse

- Für die Aufwandsanalyse wird als Beispiel eine abstrakte Methode $LV(x)$ zur Lösung des Problems x genommen
- Ergebnis von (y, Erfolg)
  - *Erfolg* nimmt einen boolean Wert an und zeigt, ob die Berechnung erfolgreich war
  - y ist im Erfolgsfall das gesuchte Resultat

Las-Vegas-Aufwandsanalyse

- Für die Aufwandsanalyse wird als Beispiel eine abstrakte Methode $LV(x)$ zur Lösung des Problems x genommen
- Ergebnis von (y, Erfolg)
  - *Erfolg* nimmt einen boolean Wert an und zeigt, ob die Berechnung erfolgreich war
  - y ist im Erfolgsfall das gesuchte Resultat
- Erfolgswahrscheinlichkeit $p(x)$, wobei $0 < p(x) < 1$

Las-Vegas-Aufwandsanalyse

Ergebnis von $LV(x)$ in der Form einer Liste: { y, Erfolg }

Java-Pseudocode

```
function LV-loop(x):
resultatListe[] = Resultat, boolean Wert;
Erfolg = resultatListe[1];
y = resultatListe[0];
if Erfolg == true  then
   return y;
else
   LV-loop(x);
end
```

Las-Vegas-Aufwandsanalyse

- Zwei Ausführungszeiten des Algorithmus für x

Las-Vegas-Aufwandsanalyse

- Zwei Ausführungszeiten des Algorithmus für x
- *success*(x)
  - Zeit, die der Algorithmus für erfolgreiches Resultat benötigt
- *failure*(x)
  - Zeit, die der Algorithmus benötigt, bis er feststellt, dass er zu einer Sackgasse gekommen ist

Las-Vegas-Aufwandsanalyse

- Zwei Ausführungszeiten des Algorithmus für x
- *success*(x)
  - Zeit, die der Algorithmus für erfolgreiches Resultat benötigt
- *failure*(x)
  - Zeit, die der Algorithmus benötigt, bis er feststellt, dass er zu einer Sackgasse gekommen ist

Jetzt wollen wir die gesamte Rechenzeit $t(x)$ unserer loop-Methode berechnen.

Las-Vegas-Aufwandsanalyse

1. $t(x) = p(x) \cdot succes(x) + (1 - p(x)) \cdot (failure(x) + t(x))$
2. $t(x) = succes(x) + \frac{1-p(x)}{p(x)} \cdot failure(x)$
3. $t(x) = succes(x) + \left(\frac{1}{p(x)} - 1 \right) \cdot failure(x)$

Fazit der Las-Vegas-Aufwandsanalyse

$$t(x) = succes(x) + \left(\frac{1}{p(x)} - 1 \right) \cdot failure(x)$$

- Mit dem steigenden Wert von $p(x)$ ist die Rechenzeit für ein richtiges Resultat normalerweise kleiner

Fazit der Las-Vegas-Aufwandsanalyse

$$t(x) = succes(x) + \left(\frac{1}{p(x)} - 1 \right) \cdot failure(x)$$

- Mit dem steigenden Wert von $p(x)$ ist die Rechenzeit für ein richtiges Resultat normalerweise kleiner
- Nur steigt aber oftmals mit dem Wert $p(x)$ auch der Wert von $failure(x)$

Fazit der Las-Vegas-Aufwandsanalyse

$$t(x) = succes(x) + \left(\frac{1}{p(x)} - 1 \right) \cdot failure(x)$$

- Mit dem steigenden Wert von $p(x)$ ist die Rechenzeit für ein richtiges Resultat normalerweise kleiner
- Nur steigt aber oftmals mit dem Wert $p(x)$ auch der Wert von $failure(x)$
- Hier kann passieren, dass Algorithmen mit höherer Erfolgswahrscheinlichkeit mehr Zeit brauchen um ein rightiges Resultat zu liefern, als die, die mehr Wiederholungen brauchen
- $p(x)$ und $failure(x)$ in Balanz halten um minimale $t(x)$ zu kriegen

Las-Vegas-Algorithmen

Sherwood-Algorithmen

Sherwood Algorithmen sind eine spezielle Form von Las-Vegas-Algorithmen.
Sie liefern auch immer ein richtiges Ergebnis, aber kommen niemals in eine Sackgasse.
Sherwood-Algorithmen sind aus deterministischen Verfahren konstruiert, die den mittleren Aufwand riesig kleiner als den schlechtesten haben.
Durch Zufallszahlen kann diese Amplitude zwischen dem mittleren und schlechtesten Aufwand balanciert.

Las-Vegas-Algorithmen

Robin-Hood-Efekt

Durch die Balancierung mit Zufallszahlen erreichen wir ein Zustand, wo sich der Algorithmus als sein deterministischer Original verhält mit dem Vorteil, dass er die gutartige, aber auch ungünstige Eingaben mit dem mittleren Aufwand liefert.
Vom besten und schlechtesten Aufwand ist genommen und dem mittleren Aufwand ist gegeben.

Monte-Carlo-Algorithmen

Monte-Carlo-Algorithmen sind Algorithmen, die ein korrektes Ergebnis liefern, obwohl s die Ergebnise bei mehreren Aufrufen unterschiedlich sein können, wie bei der Berechnung der Näherung von π.
Monte-Carlo-Algorithmen sind **nichtdeterministische Algorithmen**.

Fehler Interpretation

Bei den Monte-Carlo Algorithmen gelten 2 Aussagen:

- Wenn etwas falsch im Algorithmus läuft, wird die Antwort falsch sein. Es ist aber zu beachten, dass die Lösung unbekannt ist und deshalb kann das Ergebnis mit der allgemeine Lösung nicht vergleicht werden.
- Das bedeutet nicht, dass der Algorithmus immer richtig funktioniert.

Die Wahrscheinlichkeit einer falschen Ausgabe von einem Monte-Carlo Algorithmus ist so gering, dass es immer als richtig genommen ist.

Multimengen

Multimengen

Mengen, wo die Elementen sich wiederholen können, sind Multimengen.
Beispiel: $\{\mathbf{1},2,3,4,\mathbf{1},5\}$ ist eine Multimenge, denn das Element $\mathbf{1}$ kommt mehrmals vor.

Äquivalenz

Zwei Multimengen sind äquivalent, wenn sie die gleiche Elemente besitzen, wobei die Reihenfolge unwichtig ist.
Beispiel: $\{1,2,1\} \equiv \{2,1,1\} \not\equiv \{1,2,2\}$

Äquivalenz von Multimengen

Es wurde angenommen, dass die Multimengen die gleiche Kardinalität haben, sonst wären sie eben nicht äquivalent.
Bei der Sortierung und beim Vergleich ist es sehr aufwendig $\mathcal{O}(n \log n)$.
Mit Zufall gibt es einen linearen Aufwand von $\Theta(n)$
.

[0] Θ heißt dass der Best Case und Worst Case gleich sind

Also?

Ein Polynom wird mit einer eingegebene Zahl und der Multimenge gemacht:
$p1(x) = (x - x_1)(x - x_2)..(x - x_n)$
und gleich für die zweite Multimenge:
$p2(x) = (x - y_1)(x - y_2)..(x - y_n)$
Jedes Polynom hat dann einen Wert, wenn der Vergleich von diesen Werten aussagt, dass die Mengen äquivalent sind.

Anwendung

Vergleichen wir die Multimenge $M1\{1,2,1\}$ $M2\{2,1,1\}$ $M3\{1,2,2\}$ **mit x=2**

$$p1(3) = (2-1)\cdot(2-2)\cdot(2-1) = 0$$
$$p2(3) = (2-2)\cdot(2-1)\cdot(2-1) = 0$$
$$p3(3) = (2-1)\cdot(2-2)\cdot(2-2) = 0$$

Die Werte allen Mengen sind gleich.

[0] Wenn der Polynomwert **0** ist, heißt das, dass dieses x sich in der Menge befindet.

Anwendung

Jetzt mit **mit x=3**.

$$p1(3) = (3-1)\cdot(3-2)\cdot(3-1) = 4$$
$$p2(3) = (3-2)\cdot(3-1)\cdot(3-1) = 4$$
$$p3(3) = (3-1)\cdot(3-2)\cdot(3-2) = 2$$

Da die Werte für $p1$ und $p2$ gleich sind und der Wert von $p3$ anderes ist, sind nur $p1$ und $p2$ äquivalent.

Anwendung

Bei der *zufällige* Auswahl von x und einer bestimmte Anzahl von Wiederholungen dieser Vergleichmethode, kann man sicher sein, dass das Ergebnis richtig ist. Empfohlen ist, eine kleine Zufallszahl zu nehmen, und merhfach wiederholen.

Allgemein

- Eine Primzahl ist eine Zahl, dass nur mit 1 und sich selbst teilbar ist

Allgemein

- Eine Primzahl ist eine Zahl, dass nur mit 1 und sich selbst teilbar ist
- Durch Probeteiler kann man bestimmen, ob eine Zahl Primzahl ist

Allgemein

- Eine Primzahl ist eine Zahl, dass nur mit 1 und sich selbst teilbar ist
- Durch Probeteiler kann man bestimmen, ob eine Zahl Primzahl ist
- Bei Zahlen mit größer Stelligkeit ist diese Form sehr Zeitaufwändig

Allgemein

- Eine Primzahl ist eine Zahl, dass nur mit 1 und sich selbst teilbar ist
- Durch Probeteiler kann man bestimmen, ob eine Zahl Primzahl ist
- Bei Zahlen mit größer Stelligkeit ist diese Form sehr Zeitaufwändig
- Effizientere Lösung durch Einfügen von Zufallszahlen und Umwandlung zu einem Monte-Carlo Algorithmus mit der Rechenzeit von $\mathcal{O}(\log n)$

Satz von EULER

- Um die theoretische Basis eines solchen Algorithmus zu machen, brauchen wir den Satz von EULER

Satz von EULER

- Um die theoretische Basis eines solchen Algorithmus zu machen, brauchen wir den Satz von EULER
- Für alle natürliche Zahlen n und a mit $n \geq 2$ und $1 \leq a < n$, wobei $ggt(a, n) = 1$ gilt $a^{\phi(n)} \equiv 1 \ mod \ n$

Satz von EULER

- Um die theoretische Basis eines solchen Algorithmus zu machen, brauchen wir den Satz von EULER
- Für alle natürliche Zahlen n und a mit $n \geq 2$ und $1 \leq a < n$, wobei $ggt(a, n) = 1$ gilt $a^{\phi(n)} \equiv 1 \ mod \ n$
- Auf diesen Satz kann man den "kleinen Satz von FERMAT" anwenden

Kleiner Satz von FERMAT

- Wenn n eine Primzahl ist, dann gilt für alle natürlichen Zahlen a mit $(1 \leq a \leq n-1)$:

Kleiner Satz von FERMAT

- Wenn n eine Primzahl ist, dann gilt für alle natürlichen Zahlen a mit $(1 \leq a \leq n-1)$:
  - $a^n \equiv a \ mod \ n$

Kleiner Satz von FERMAT

- Wenn n eine Primzahl ist, dann gilt für alle natürlichen Zahlen a mit $(1 \leq a \leq n-1)$:
  - $a^n \equiv a \ mod \ n$
- Aus beiden Sätzen wird dann: $a^{n-1} \equiv 1 \ mod \ n$

Kleiner Satz von FERMAT

- Wenn n eine Primzahl ist, dann gilt für alle natürlichen Zahlen a mit $(1 \leq a \leq n-1)$:
  - $a^n \equiv a \ mod \ n$
- Aus beiden Sätzen wird dann: $a^{n-1} \equiv 1 \ mod \ n$
- Bei diesem Primzahltest nach FERMAT" gibt es aber Zahlen, die den Test erfüllen, sind aber trotzdem nicht Primzahlen

Kleiner Satz von FERMAT

- Wenn n eine Primzahl ist, dann gilt für alle natürlichen Zahlen a mit $(1 \leq a \leq n-1)$:
  - $a^n \equiv a \ mod \ n$
- Aus beiden Sätzen wird dann: $a^{n-1} \equiv 1 \ mod \ n$
- Bei diesem Primzahltest nach FERMAT" gibt es aber Zahlen, die den Test erfüllen, sind aber trotzdem nicht Primzahlen
- **Pseudoprimzahlen** und **Carmichael-Zahlen**

Pseudoprimzahlen

- Pseudoprimzahlen sind zusammengesetzte Zahlen n, zu den es mindestens ein a mit $ggT(a, n) = 1$ und $a > 1$ gibt

Pseudoprimzahlen

- Pseudoprimzahlen sind zusammengesetzte Zahlen n, zu den es mindestens ein a mit $ggT(a, n) = 1$ und $a > 1$ gibt
- Dabei gilt dann auch $a^{n-1} \equiv 1 \ mod \ n$

Pseudoprimzahlen

- Pseudoprimzahlen sind zusammengesetzte Zahlen n, zu den es mindestens ein a mit $ggT(a, n) = 1$ und $a > 1$ gibt
- Dabei gilt dann auch $a^{n-1} \equiv 1 \ mod \ n$
- z.B. für die Zahl $341 = 11 \cdot 31$ gilt die Kongruenz $2^{340} \equiv 1 \ mod \ 341$

Pseudoprimzahlen

- Pseudoprimzahlen sind zusammengesetzte Zahlen n, zu den es mindestens ein a mit $ggT(a, n) = 1$ und $a > 1$ gibt
- Dabei gilt dann auch $a^{n-1} \equiv 1 \ mod \ n$
- z.B. für die Zahl $341 = 11 \cdot 31$ gilt die Kongruenz $2^{340} \equiv 1 \ mod \ 341$
- 341 ist eine Pseudoprimzahl bezüglich der Basis 2 , aber nicht bezüglich der Basis 3

Pseudoprimzahlen

- Pseudoprimzahlen sind zusammengesetzte Zahlen n, zu den es mindestens ein a mit $ggT(a, n) = 1$ und $a > 1$ gibt
- Dabei gilt dann auch $a^{n-1} \equiv 1 \ mod \ n$
- z.B. für die Zahl $341 = 11 \cdot 31$ gilt die Kongruenz $2^{340} \equiv 1 \ mod \ 341$
- 341 ist eine Pseudoprimzahl bezüglich der Basis 2 , aber nicht bezüglich der Basis 3
- Basis 2 ist ein falscher Zeuge(Witness) für den Primzahlcharakter 341

Carmichael-Zahlen

- Eine zusammengesetzte Zahl n ist Carmichael-Zahl, falls für alle zu n teilerfremden Zahlen a die folgende Kongruenz erfüllt ist:
  $a^{n-1} \equiv 1 \ mod \ n$

Carmichael-Zahlen

- Eine zusammengesetzte Zahl n ist Carmichael-Zahl, falls für alle zu n teilerfremden Zahlen a die folgende Kongruenz erfüllt ist:
  $a^{n-1} \equiv 1 \ mod \ n$
- Kleinste Carmichael-Zahl ist 561. $561 = 3 \cdot 11 \cdot 17$

Carmichael-Zahlen

- Eine zusammengesetzte Zahl n ist Carmichael-Zahl, falls für alle zu n teilerfremden Zahlen a die folgende Kongruenz erfüllt ist: $a^{n-1} \equiv 1 \ mod \ n$
- Kleinste Carmichael-Zahl ist 561. $561 = 3 \cdot 11 \cdot 17$
- 561 ist mit 3,11,17,33,51,187 teilbar.

Probabilistischer Primzahltest

- Beim probabilistischen Primzahltest wird die Basis zufällig gewählt
- Dadurch wird der Zeitaufwand sehr stark verbessert, aber mit möglichkeit von falschen Ergebnisen
- Mehrmalige Ausführung verbessert die Schanz, richtige Ergebnise zu bekommen

Jacobi-Funktion

- In 1977 entwickelt, benutzt bei der Kryptographie

Jacobi-Funktion

- In 1977 entwickelt, benutzt bei der Kryptographie
- In diesem Test wird erst zufällig ein $a < n$ mit $ggt(a, n) = 1$ gewählt

Jacobi-Funktion

- In 1977 entwickelt, benutzt bei der Kryptographie
- In diesem Test wird erst zufällig ein $a < n$ mit $ggt(a, n) = 1$ gewählt
- Dann wird *Jacobi-Funktion* berechnet

$$J(a,n) = \begin{cases} 1 & , falls\ a = 1 \\ J(\frac{a}{2}, n) \cdot (-1) \cdot \frac{n^2-1}{8} & , falls\ a = gerade \\ J(rest(a,n) \cdot (-1) \cdot (a-1)^{\frac{n-1}{4}} & , sonst \end{cases}$$

$$J(a,n) = \begin{cases} 1 & , falls\ a = 1 \\ J(\frac{a}{2}, n) \cdot (-1) \cdot \frac{n^2-1}{8} & , falls\ a = gerade \\ J(rest(a,n) \cdot (-1) \cdot (a-1)^{\frac{n-1}{4}} & , sonst \end{cases}$$

- $J(a,n)$ kann nur die Werte $+1$ und -1 annehmen. Es gilt dann folgender Satz:

$$J(a,n) = \begin{cases} 1 & , falls\ a = 1 \\ J(\frac{a}{2}, n) \cdot (-1) \cdot \frac{n^2-1}{8} & , falls\ a = gerade \\ J(rest(a,n) \cdot (-1) \cdot (a-1)^{\frac{n-1}{4}} & , sonst \end{cases}$$

- $J(a,n)$ kann nur die Werte $+1$ und -1 annehmen. Es gilt dann folgender Satz:
- Wenn n eine Primzahl ist, dann gilt für alle $a \in N$ mit $1 \leq a \leq n-1$ und $ggT(a,n) = 1$:
  $J(a,n) \equiv a^{\frac{n-1}{2}}\ mod\ n$

Ergebnis

- Falls n ein Primzahl ist, wird immer ein korrektes Ergebnis vom Test geliefert

Ergebnis

- Falls n ein Primzahl ist, wird immer ein korrektes Ergebnis vom Test geliefert
- Falls n kein Primzahl ist, liefert der Test mit der Wahrscheinlichkeit $< \frac{1}{2}$ ein falsches Ergebnis

Ergebnis

- Falls n ein Primzahl ist, wird immer ein korrektes Ergebnis vom Test geliefert
- Falls n kein Primzahl ist, liefert der Test mit der Wahrscheinlichkeit $< \frac{1}{2}$ ein falsches Ergebnis
- Durch wiederholte Ausführung des Tests mit der Anzahl k von Wiederholungen wird die Wahrscheinlichkeit auf $\frac{1}{2^k}$ reduziert

Q&A

Fragen?