Randomisierte Algorithmen

Tobias Rees,
Seraya Takahashi,
Marco Wettstein

Deterministische Algorithmen

Vorteile

- korrekter Output
- bei gleichem Input sind Laufzeit/Anzahl Steps und Output immer gleich

Nachteile

- gewisse Probleme lassen sich nicht in akzeptabler Laufzeit lösen

  → Lösung: ***randomisierte Algorithmen***

Randomisierte Algorithmen

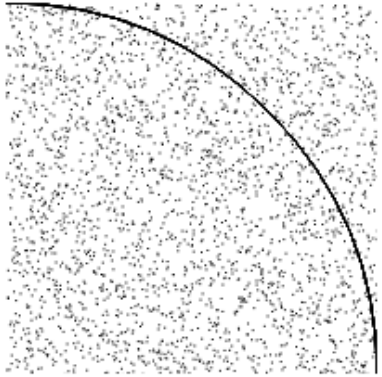
- Erweiterung des Inputs mit einer Menge von Zufallszahlen
- Ergebnis kann bei gleichem Input bei mehrmaliger Ausführung des Algorithmus variieren:
  - Laufzeit kann varieren (“*Las-Vegas*”-Algorithmen)
  - Wahrscheinlichkeit für korrektes Ergebnis kann varieren (“*Monte-Carlo*”-Algorithmen)

Randomisierte Algoritmen- Einführendes Beispiel

- Monte Carlo Simulation
- Pi ausrechnen: Buffonsches Nadelproblem → Experiment

```
approx_pi = (n) ->
    inside = 0
    for i in [1..n]
        x = Math.random()
        y = Math.random()
        if x*x+y*y <=1
            inside++
    return 4 * inside / n
```

steps:
2852
Pi:
3.1248246844319776



Vorteile 👍

- i.d.R. leicht verständlich
- meist simpel zu implementieren
- können erheblich effizienter sein als deterministische Algorithmen
- Worst- und Bestcase-Laufzeiten können “geglättet” bzw. vom Input unabhängig gemacht werden (z.B. Random Quicksort)
- Bei Physikalischen Simulationen werden häufig statistisch zufällig verteilte Anfangssituationen benötigt.
- manchmal einzige Möglichkeit, gewisse Probleme zu lösen

Nachteile 👎

- Output kann falsch sein (*“Monte Carlo”*)
- Laufzeit kann stark variieren (*“Las Vegas”*)
- Laufzeit oder Wahrscheinlichkeit eines korrekten Outputs sind i.d.R. schwierig festzustellen
- garantiert “zufällige” Zahlen zu erhalten ist unmöglich (Abhängigkeit von angegebener Menge Pseudo-Zufallszahlen)
  → Resultat abhängig von “Qualität” der Pseudo-Zufallszahlen


Las Vegas
Algorithmen
"Always right,
probably fast"

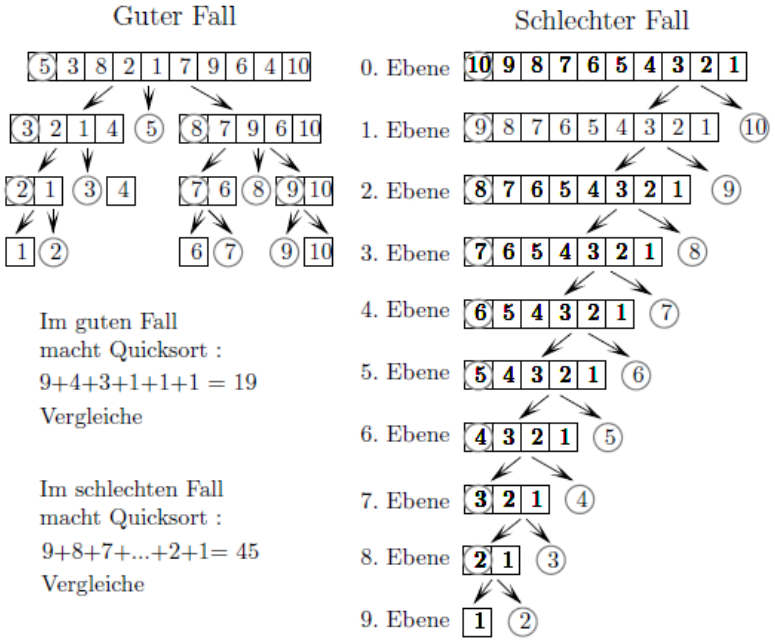
Las Vegas

- liefert nie falsche Ergebnisse
- 2 Definitionen
  - ein Algorithmus, der immer das richtige Resultat liefert, **falls** er terminiert (Zeitkomplexität abhängig von Zufallsvariable)
  - ein Algorithmus, der das richtige Resultat liefert (WSK >= 50%) oder keines liefert (WSK <= 50%)
- Komplexitätsklassen
  - ZPP - *zero error probabilistic polynomial*

$w \in L$	Prob(M(w) = 0) = 0	Prob(M(w) = 1) > 1/2
$\neg(w \in L)$	Prob(M(w) = 1) = 0	Prob(M(w) = 0) > 1/2

Determistischer Quicksort

- Worst Case: sortierte Liste



Random Quicksort: Algorithmus

RAND-QUICKSORT(A, i, j)

Input: Ein Array A[1 . . . n] und zwei Indizes 1 <= i <= j <= n

Output: Das Teilfeld A[i . . . j] wird aufsteigend sortiert.

1. **Wähle x aus A[i . . . j] zufällig und gleichverteilt.**
2. Teile A[i . . . j] durch Vergleich mit dem Pivotelement x auf in die Elemente A[i . . . k - 1] kleiner als x, A[k] = x und die Elemente A[k + 1 . . . j] größer als x.
3. if i < k - 1 then
4. QUICKSORT(A, i, k - 1)
5. end if
6. if k + 1 < j then
7. QUICKSORT(A, k + 1, j)
8. end if

Random Quicksort: WSK Pivotelement

- wählt ein zufälliges Pivotelement in jedem Step
- Anzahl Vergleiche ist im Schnitt 2n*Hn
- Hn ist die nt Zahl der Harmonische Reihe und kann auch als ln(n) + 1 dargestellt werden

$$H_n = \sum_{k=1}^{n} \frac{1}{k} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \leq ln(n) + 1$$

- darum hat 2n*Hn eine durchschnittliche Laufzeitkomplexität von O(n*log (n))
- und ist damit gleich wie der det. Quicksort
- Aber unabhängig vom Input, ob sortiert oder unsortiert

Random Quicksort - Experiment

→ Experiment

Graphen Isomorphie Problem

- Zwei Graphen auf Isomorphie (Gleichheit) testen
- üblicherweise mit Backtracking-Algorithmen gelöst (Random Search)
- Beispielalgorithmen: Ullman, VF, VF2

- Anwendung:
  - molekularer Graphen
  - Algorithmische Biologie
  - Bildanalyse und -verarbeitung
  - Musterkennung
  - Graphgrammatiken, Graphtransformationen

Monte Carlo Algorithmen

"Always fast, probably right"

Eigenschaften Monte-Carlo

- kann falsche Ergebnisse liefern
- Qualität gemessen in oberer Schranke der Fehlerwahrscheinlichkeit
- Komplexitätsklassen, M ist (randomisierte) Turing Maschine

Class	**Wahrscheinlichkeit**	**Beschreibung**	**Fehler**
PP	Prob(M(w) = L(w)) > ½	probalistic polonomial	beidseitig
BPP	Prob(M(w) = L(w)) > ½ + *e* mit *e* > 0	bounded error probalistic polonomial	beidseitig
RP ∧ w∈L **RP ∧ ¬(w∈L)**	Prob(M(w) = 1) > ½ Prob(M(w) = 0) = 1	random polynomial	einseitig

Fehlerarten für Entscheidungsprobleme

Erlaubte Kombinationen

Zweiseitige Fehler

Kompl.Kl.: PP, BPP

L(w) richtige Lösung	M(w) Ausgabe
w	f
w	w
f	f
f	w

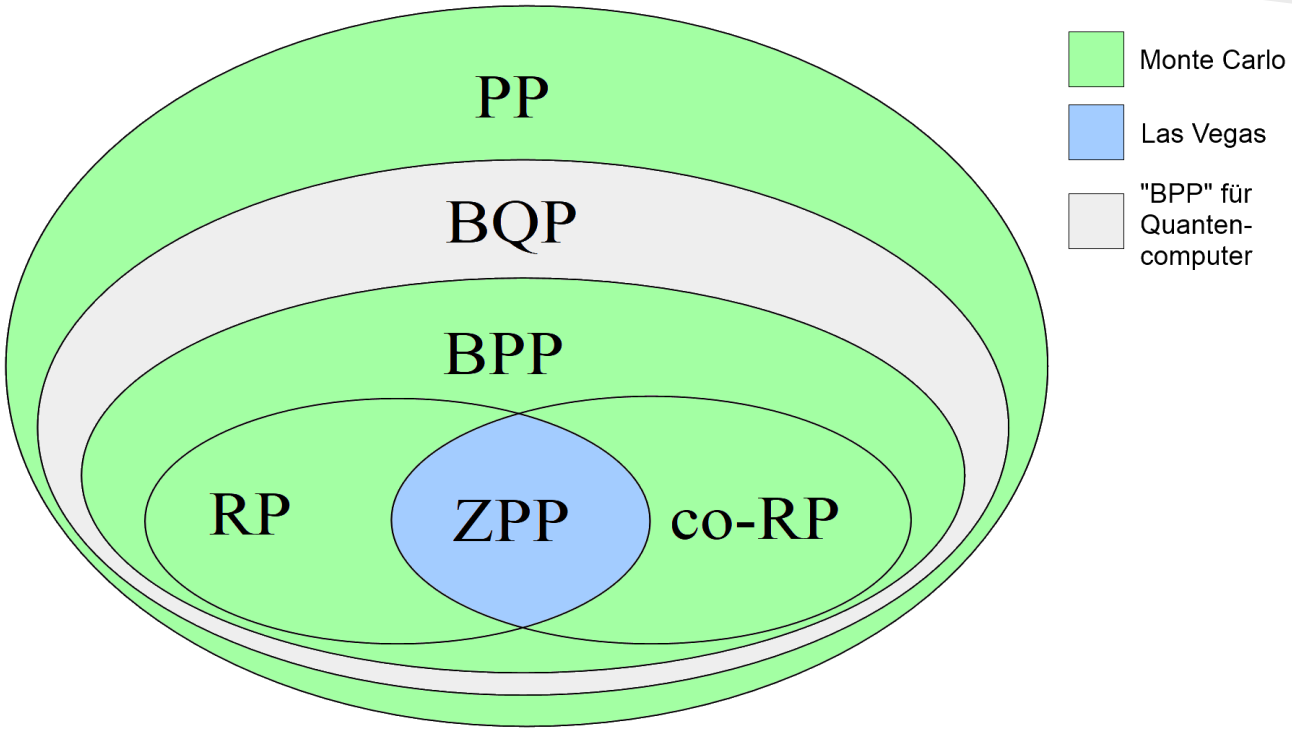
Einseitige Fehler

Kompl.Kl.: RP

L(w) richtige Lösung	M(w) Ausgabe
w	f
w	w
f	f
~~f~~	~~w~~

nicht erlaubt!

Komplexitätsklassen

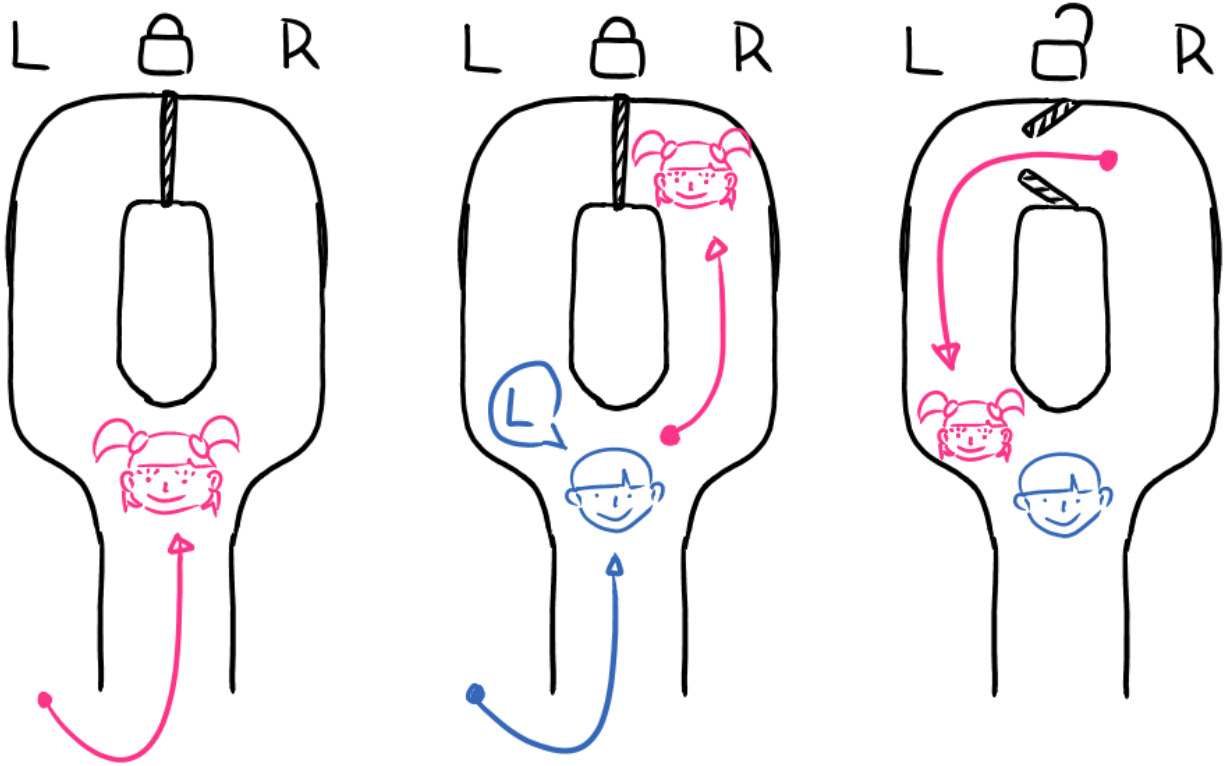


Monte Carlo - Beispiele

Zero Knowledge Proof

- Randomisiertes (*Monte-Carlo*-) Protokoll
- 2 Parteien
  - Beweiserin (Alice)
  - Verifizierer (Bob)
- Beweiserin überzeugt Verifizierer, dass sie das Geheimnis kennt, ohne Informationen über das Geheimnis selbst herauszugeben
- Verifizierer erlangt kein neues Wissen
- Pro erfolgreichem Durchlauf sinkt die Wahrscheinlichkeit, dass Alice das Geheimnis nicht kennt

Zero Knowledge Proof



Wahrschein-
lichkeit, dass A das
Geheimnis kennt
nach *n* Tests:

$$1 - 2^{-n}$$

Primzahlentests

- Algorithmus testet, ob eine gegebene Zahl eine Primzahl ist oder nicht
- Ausgabe:

$M(n) = 1 \rightarrow$ n ist Primzahl

$M(n) = 0 \rightarrow$ n ist keine Primzahl

- Werden für Kryptographie gebraucht, dabei werden grosse Primzahlen für die Schlüsselerstellung gebraucht
- Deterministische Algorithmen i.d.R. ineffizient:
    - häufig exponentionelle Laufzeit
    - AKS-Primzahltest zwar polynomiell, aber mit hohen Potenzen

Primzahlentests

Lösung: Randomisierte Algorithmen:

- Solovay-Strassen-Test
- **Miller-Rabin**-Test

→ Liefern nur mit gewisser Wahrscheinlichkeit korrektes Ergebnis.

→ Wiederholung des Tests reduziert Irrtumswahrscheinlichkeit

→ sind daher *Monte-Carlo*-Algorithmen

Primzahlentest - Miller-Rabin-Test

- Eingabe:
    - natürliche, ungerade Zahl n
    - k Schritte
- Ausgabe nach k Schritten:
    - n ist keine Primzahl
    - n ist wahrscheinlich eine Primzahl (oder starke Pseudoprimzahl)
- Wahrscheinlichkeit für Irrtum nach k Schritten: $\frac{1}{4^k}$

→ Ist *Monte-Carlo*-Algorithmus

Primzahlentest - Miller-Rabin-Test

Idee: Es existieren Tests, die nur Primzahlen oder “starke Pseudoprimzahlen” bestehen:

- Man wählt in jedem Durchgang zufällig einen “Zeugen” a und führt diesen Test aus.
- Falls ein Test bestanden steigt Wahrscheinlichkeit eine Primzahl zu haben um Faktor ¾.
- Wird ein Test aber nicht bestanden, handelt es sich in jedem Fall um *keine* Primzahl. → Einseitiger Fehler

→ Experiment

Derandomisierung

- Menge der Zufallsvariablen verringern
- Motivation: Durch “ausprobieren” aller Belegungen der Zufallsvariablen kann ein Randomisierter Algorithmus deterministisch gemacht werden
- Bei c Zufallsvariablen steigt Rechenzeit um Faktor 2^c
  → führt bei “naivem” Ansatz schnell zu exponentioneller Laufzeit
- Falls c aber klein, z.b. falls für Eingabelänge n gilt: c = logn
  → Laufzeit steigt nur polynomiell

  → *Daher möglichst wenig Zufallsvariablen*

Derandomisierung - Deterministischer Miller-Rabin

- Idee: statt Basis a in jedem Durchgang zufällig zu wählen, alle a durchprobieren.

- Durch Wahl “geeigneter” a kann ein sehr effizienter, deterministischer Miller-Rabin für gewisse Eingabelängen n erzeugt werden.

- Falls *“Riemannsche Vermutung”* wahr gilt für die Laufzeit: $O((\log n)^4)$

Derandomisierung - Deterministischer Miller-Rabin

Bei kleinen n reichen sogar sehr wenige Basen:

n ist kleiner als	**zu testende Basen a**
1.373.653	2, 3
9.080.191	31, 73
4.759.123.141	2, 7, 61
2.152.302.898.747	2, 3, 5, 7, 11
3.474.749.660.383	2, 3, 5, 7, 11, 13
341.550.071.728.321	2, 3, 5, 7, 11, 13, 17

→ siehe auch http://miller-rabin.appspot.com/

Quellen und Weblinks

- Online-Demos für die Präsentation: http://random.macrozone.ch/
- github von unserem code: https://github.com/macrozone/random_alg
- Dienstleistungen rund um Zufallszahlen: http://www.random.org/
- geeignete Basen für den deterministischen Miller-Rabin-Test: http://miller-rabin.appspot.com/
- Primzahlen: http://primes.utm.edu/
- Miller Rabin: http://de.wikipedia.org/wiki/Miller-Rabin-Test
- Miller Rabin als Pseudocode erklärt: http://stackoverflow.com/a/17078819/1463534
- allgemeines über Randomisierte Algoritmen: http://de.wikipedia.org/wiki/Randomisierter_Algorithmus
- “Ali Baba und die 40 Räuber”, Zero-Knowledge-Proof anschaulich erklärt: http://pages.cs.wisc.edu/~mkowalcz/628.pdf
- Randomisierte Algorithmen & Probabilistische Analyse, TU Berlin http://optimierung.mathematik.uni-kl.de/~krumke/Notes/rand-alg-skript.pdf
- Vorlesung ‘Randomisierte Algorithmen’, Universität Karlsruhe http://liinwww.ira.uka.de/~thw/vl-rand-alg/

Quellen und Weblinks

- Randomisierte Algorithmen von Sabrina Wiedersheim, ETHZ Arbeit
  http://www.abz.inf.ethz.
  ch/abz/media/archive1/unterrichtsmaterialien/maturitaetsschulen/Rand-Alg.pdf
- Beispielprogramm von Graph Isomorphism Algorithm
  http://www.dharwadker.org/tevet/isomorphism/
- Las Vegas Algorithmen für GIP:
  http://ceit.aut.ac.ir/~meybodi/paper/beigy-meybodi-Las-vegas-Graph%20isomorphism.ps