Algorithmen in der Biologie

Dr. Hans-Joachim Böckenhauer
Dr. Dennis Komm

Zusammenfassung des 1. Abends

Zürich, 23. April 2014

1 Grundlagen der Molekularbiologie

Für eine Einführung in die Grundbegriffe der Molekularbiologie, wie wir sie in diesem Kurs brauchen, siehe das Kapitel 2 im Buch „Algorithmische Grundlagen der Bioinformatik“ von Böckenhauer und Bongartz [1].

2 Einführung in die Algorithmik

Die Informatik ermöglicht eine neue Sichtweise auf komplexe biologische Probleme. *Algorithmisches Denken* hilft bei der Formulierung und Lösung vieler Forschungsaufgaben. In diesem Kurs wollen wir einige Grundprinzipien des algorithmischen Denkens am Beispiel biologischer Anwendungen kennenlernen.

Die Lösung eines biologischen Problems mit Hilfe algorithmischer Methoden erfordert zunächst eine *Modellierung* der biologischen Fragestellung in der „Sprache“ der Informatik. Das Vorgehen ist schematisch in Abbildung 1 gezeigt.

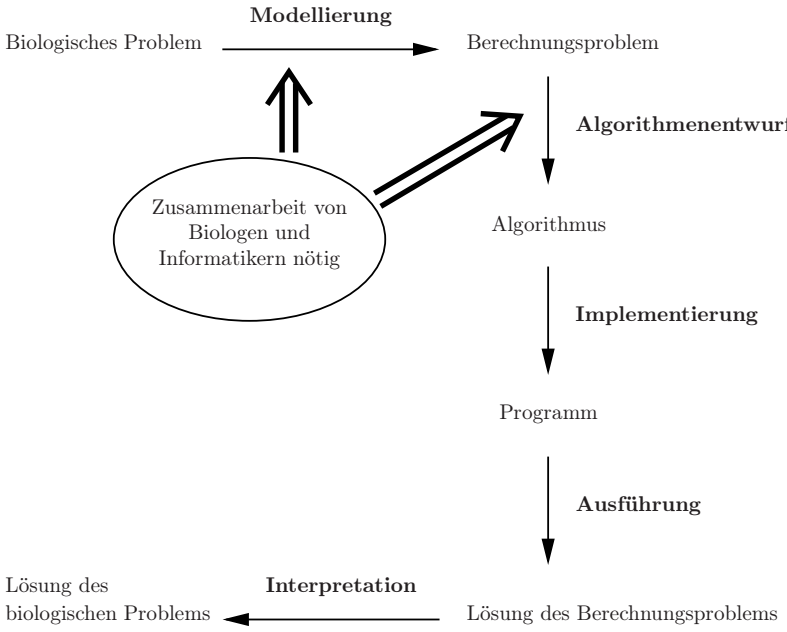

Abbildung 1. Modellierung und Algorithmenentwurf zur Lösung von biologischen Problemen

Ein *Algorithmus* ist ein systematisches Verfahren zur Lösung eines Berechnungsproblems. Er besteht aus einer Folge „einfacher“ Anweisungen in festgelegter Reihenfolge, ist endlich beschreibbar und berechnet *immer* eine korrekte Lösung, insbesondere *terminiert* er immer,

beendet also für jede Eingabe nach endlicher Zeit die Berechnung. Um einen Algorithmus beschreiben zu können, müssen wir natürlich zunächst das Problem, das dieser lösen soll, formalisieren. Wir verlassen an dieser Stelle einmal ganz kurz die Biologie und betrachten das Problem, Primzahlen zu erkennen. Dieses ist in der Praxis, etwa bei der Kommunikation oder dem Online-Banking, von enormer Bedeutung.

2.1 Ein Primzahltest

Die Aufgabe ist, für eine gegebene Zahl zu entscheiden, ob sie eine Primzahl ist oder nicht. Was heisst das genau? Nichts anderes als zu prüfen, ob die gegebene Zahl von einer anderen (ausser der Eins) geteilt wird. Es geht hier also genau um die in Abbildung 1 beschriebene Modellierung durch ein Berechnungsproblem.

> **Eingabe:** Eine beliebige Zahl
> **Ausgabe:** „Primzahl“ oder „Keine Primzahl“ je nachdem, ob diese Zahl eine Primzahl ist.

Unsere formale Problembeschreibung gibt uns also vor, welche Eingabe und welche Ausgabe ein Primzahlalgorithmus erwartet beziehungsweise produziert. Wie können wir diesen nun formulieren? Wenn wir herausfinden wollen, ob irgendeine kleinere Zahl ausser 1 die gegebene teilt, so können wir dies natürlich einfach für alle ausprobieren. Bezeichnen wir die Eingabe mit x, wobei wir also immer annehmen, dass x eine natürliche Zahl grösser 2 ist (also 3, 4, 5, ...). Dann sieht unser Vorgehen in Worten wie folgt aus.

> **Eingabe:** Eine beliebige natürliche Zahl x
> **Teste** für alle Zahlen von 2 bis $x - 1$, ob sie x teilen
> Wurde eine solche Zahl gefunden?
> **Wenn** ja
> **Ausgabe:** „Keine Primzahl“
> **Ende**
> **Sonst**
> **Ausgabe:** „Primzahl“
> **Ende**

Um dieses Vorgehen für den Computer verständlich zu machen, müssen wir die Anweisungen noch ein bisher konkreter machen. Beispielsweise ist für die meisten Menschen unmittelbar aus dem Kontext klar, was mit der Frage „Wurde eine solche Zahl gefunden?“ gemeint ist, aber für einen Computer muss dies eindeutiger formuliert werden. Klarere Anweisungen können wir beispielsweise mit folgenden Schritten formalisieren.

- Teilt 2 die gegebene Zahl x? Falls ja, **Ausgabe:** „Keine Primzahl“ **Ende**
- Teilt 3 die gegebene Zahl x? Falls ja, **Ausgabe:** „Keine Primzahl“ **Ende**
- Teilt 4 die gegebene Zahl x? Falls ja, **Ausgabe:** „Keine Primzahl“ **Ende**

  ⋮

- Teilt $x - 1$ die gegebene Zahl x? Falls ja, **Ausgabe:** „Keine Primzahl“ **Ende**
- **Ausgabe:** „Primzahl“ **Ende**

Damit wir nicht für jedes x einen Algorithmus schreiben müssen, sondern einen Algorithmus haben, der für alle x funktioniert, benutzen wir hierzu eine Variable y, die nacheinander alle Werte zwischen 2 und $x-1$ annimmt.

Eingabe: Eine beliebige natürliche Zahl x
Setze $y = 2$.
Wiederhole, solange y kleiner als x ist, folgende 6 Schritte
1. **Teste**, ob x von y geteilt wird
2. **Wenn** ja
3. **Ausgabe:** „Keine Primzahl“
4. **Ende**
5. **Sonst**
6. **Erhöhe** y um 1

Ausgabe: „Primzahl“ (An dieser Stelle haben wir alle Zahlen getestet)
Ende

Ein solcher Algorithmus führt uns immer zum Ziel. Die Handlungsvorschrift kann von jedem Menschen, der der deutschen Sprache mächtig ist, mit ausreichender Geduld für jedes gegebene x ausgeführt werden, um zu bestimmen, ob x eine Primzahl ist. Die einzelnen „Befehle“ sind klar definiert und unzweideutig zu interpretieren. Dies ist sehr nahe an der konkreten Implementierung des Algorithmus (analog zu Abbildung 1 reden wir nun von einem *Programm*) in einer höheren Programmiersprache, die ein Computer versteht.

2.2 Laufzeit

In der Informatik geht es uns um die Entwicklung und das Studium von Algorithmen für verschiedene Probleme, die eben beispielsweise ihren Ursprung in biologischen Fragestellungen haben. Eines der Hauptkriterien, das entscheidet, inwiefern uns ein Algorithmus hilft, ist dessen *Laufzeit*. Diese hängt meistens von der Grösse der konkreten Eingabe ab. Wenn wir obiges Beispiel betrachten, so stellen wir fest, dass der Algorithmus in etwa eine Anzahl von Divisionen vornimmt, die der Grösse von x entspricht (wenn x eine Primzahl ist). Jede dieser Divisionen kostet eine gewisse Rechenzeit. Wollen wir beispielsweise feststellen, ob die Zahl 340 027 eine Primzahl ist, muss unser Algorithmus ungefähr 340 000 Divisionen durchführen.

Aus diesem Grund ist es in der Informatik oftmals ratsam, ein bisschen Mathematik zu können. Wir können die Anzahl der Tests nämlich drastisch verkleinern, wenn wir folgenden Satz benutzen.

Satz 1. *Sei x eine natürliche Zahl, die keine Primzahl ist. Dann wird x von mindestens einer Zahl geteilt, die kleiner ist als $\sqrt{x}$.*

Beweis. Sei also x die gegebene Zahl. Wenn x keine Primzahl ist, dann wird x von einer anderen natürlichen Zahl a geteilt und a ist weder 1 noch x. Wenn dies aber der Fall ist, dann muss x auch noch von einer anderen Zahl geteilt werden, die weder x noch 1 ist, nämlich von der Zahl $b = x : a$. Wir behaupten jetzt also, dass entweder a oder b höchstens $\sqrt{x}$ ist, dass es also nicht sein kann, dass beide grösser sind als $\sqrt{x}$. Dies wird uns sofort klar, wenn wir uns überlegen, dass, wenn sowohl $a > \sqrt{x}$ als auch $b > \sqrt{x}$ gilt, folgt, dass $a \cdot b > \sqrt{x} \cdot \sqrt{x} = x$ ist, was nicht sein kann, da ja gerade $a \cdot b = x$ gilt. □

Aus Satz 1 folgt sofort, dass wir nur noch zu testen brauchen, ob unsere gegebene Zahl x von einer Zahl zwischen 2 und $\sqrt{x}$ geteilt wird. Falls ja, ist sie sicher keine Primzahl.

Wird sie jedoch von keiner solchen Zahl geteilt, so wissen wir jetzt, dass sie auch von keiner anderen natürlichen Zahl geteilt wird und demnach eine Primzahl ist. Wir können unseren Algorithmus für das Testen also wie folgt ändern.

Eingabe: Eine beliebige natürliche Zahl x
Setze $y = 2$.
Wiederhole, solange y kleiner als oder gleich $\sqrt{x}$ ist, folgende 6 Schritte
1. **Teste**, ob x von y geteilt wird
2. **Falls** ja
3. **Ausgabe:** „Keine Primzahl“
4. **Ende**
5. **Sonst**
6. **Erhöhe** y um 1

Ausgabe: „Primzahl“ **Ende**

Die Konsequenz ist, dass wir nun viel weniger Divisionen machen müssen, nämlich nur noch ungefähr $\sqrt{x}$ viele anstatt x, wenn der Algorithmus eine Zahl x gegeben bekommt, die eine Primzahl ist. Für die Zahl 340 027 muss dieser Algorithmus nun nicht mehr ca. 340 000 Divisionen durchführen, sondern nur noch in etwa 800. Noch klarer wird der Unterschied bei wirklich grossen Zahlen. Für die Zahl 100 000 000 000 031 würde ein handelsüblicher Computer mit dem naiven Ansatz um die 3100 Jahre brauchen, um ein Ergebnis auszugeben. Somit ist die Berechnung nicht nur schwierig, sondern schlichtweg unmöglich. Verwenden wir jedoch Satz 1, so kann das Problem plötzlich in weniger als 3 Stunden gelöst werden. Dies ist insbesondere deswegen erstaunlich, weil der erste Ansatz auf einem Computer, der 100-mal schneller ist, noch immer 31 Jahre benötigt. Wir sehen also, dass eine geschickter Algorithmus viel wesentlicher für das effiziente Bearbeiten eines Problems ist als grössere Ressourcen.

Gerade in der Biologie, wo die Eingaben mitunter sehr gross und die Algorithmen sehr komplex werden können, ist ein derartiges cleveres Vorgehen sehr wichtig.

3 Ein erstes biologisches Anwendungsbeispiel

Unser Einführungsbeispiel hat mit Biologie noch immer nicht viel zu tun, obwohl Primzahlen auch in der Biologie eine ungeahnt wichtige Rolle spielen [4]. Nun wollen wir uns einem Problem widmen, für das wir die DNA von Bakterien untersuchen müssen.

3.1 Das Problem

Bei der Untersuchung einer Wasserprobe finden sich Bakterien im Wasser. Um die Arten der Bakterien genau zu bestimmen, will man eine DNA-Analyse machen.

3.2 Was ist die DNA?

Die DNA (Desoxyribonukleinsäure) ist ein Molekül, das in jeder Zelle eines jeden lebenden Organismus vorkommt und die Erbinformationen des Lebewesens enthält. Sie ist ein Molekül, das aus zwei langen Ketten sogenannter Nukleotide besteht, die helixförmig umeinander gewickelt sind. Es gibt vier verschiedene dieser Nukleotide, und zwei verschiedene DNA-Moleküle unterscheiden sich nur in der Abfolge dieser Nukleotide in der Kette. (Die zweite Kette ist komplementär zur ersten, also durch diese vollständig bestimmt.) Damit kann man ein DNA-Molekül vollständig beschreiben durch die Abfolge der Nukleotide, die man

in der Regel nach den Anfangsbuchstaben ihrer chemischen Bezeichnung mit `A`, `C`, `G` und `T` notiert. Ein DNA-Molekül lässt sich also kompakt beschreiben durch einen String (eine Zeichenreihe), bestehend aus den Buchstaben `A`, `C`, `G` und `T`. Dieser Weg vom Molekül zum String ist auch in Abbildung 2 dargestellt.

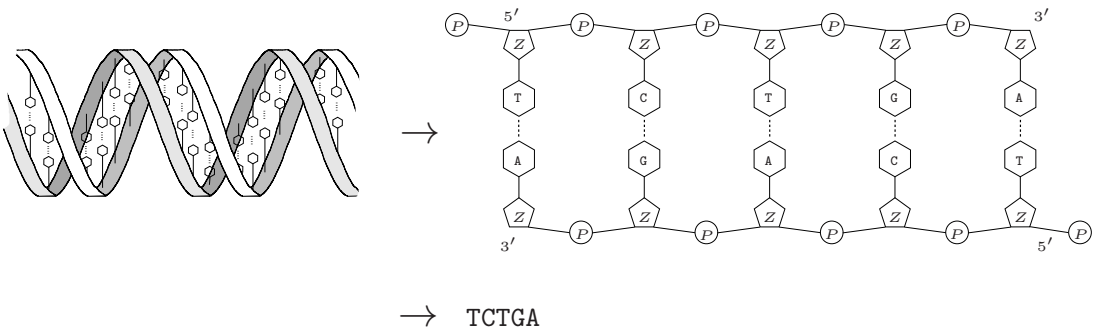

Abbildung 2. Schematische Darstellung der DNA

3.3 Das Vorgehen

Wie eine solche DNA-Analyse im Prinzip abläuft und wo man Algorithmen braucht, ist in Abbildung 3 gezeigt. Aus der Probe wird zunächst die Bakterien-DNA isoliert. Eine solche Bakterien-DNA ist etwa eine Million Nukleotide lang. Diese wird dann mit Hilfe sogenannter Restriktionsenzyme in Stücke der Grösse etwa 1 000 zerschnitten. Diese Stücke sind nun klein genug und können sequenziert werden, d. h. es wird die Folge der Nukleotide ausgelesen. Um dann festzustellen, um welches Bakterium es sich bei dieser DNA-Probe handelt, muss man nun die sequenzierten Fragmente mit den in einer Datenbank abgespeicherten Sequenzen bereits bekannter Bakteriengenome vergleichen.

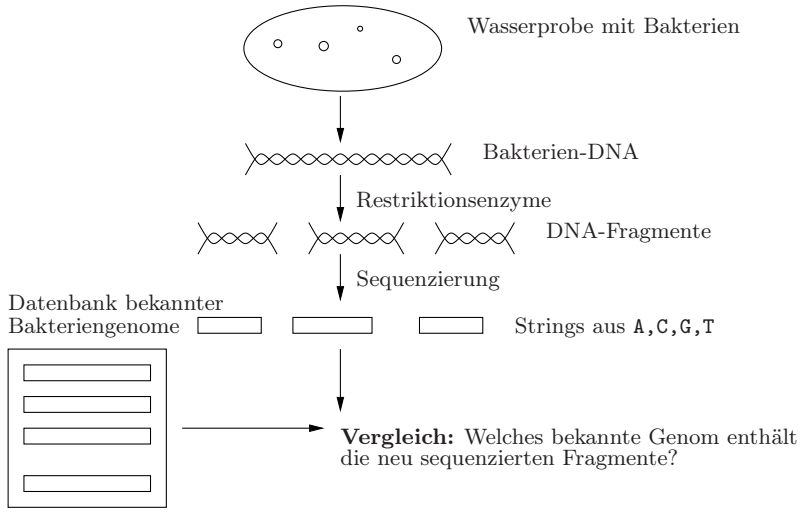

Abbildung 3. Schema der DNA-Analyse

Die Analyse führt also auf das Berechnungsproblem, einen kurzen String (das Fragment der neu sequenzierten DNA) in einem langen String (einer der Bakterien-DNAs in der Datenbank) zu vergleichen.

Eingabe: DNA-Fragment **Ausgabe:** „Nicht gefunden“ oder Position in der Datenbank

Frage: Wie kann man diese Aufgabe lösen? Wie effizient ist dies möglich?

3.4 Erster Lösungsansatz

Im Folgenden werden wir den langen String den *Text* nennen und den kurzen String das *Muster*. Wir suchen also nach einem Verfahren, das Muster in dem Text zu finden.

Der einfachste Ansatz hierfür besteht darin, das Muster mit jedem gleichlangen Teilstück des Texts zu vergleichen, siehe das Beispiel in Abbildung 4. Dort ist der Text mit t bezeichnet, das Muster mit m, und j ist die jeweilige Position in dem Text, von der ab das Muster gesucht wird.

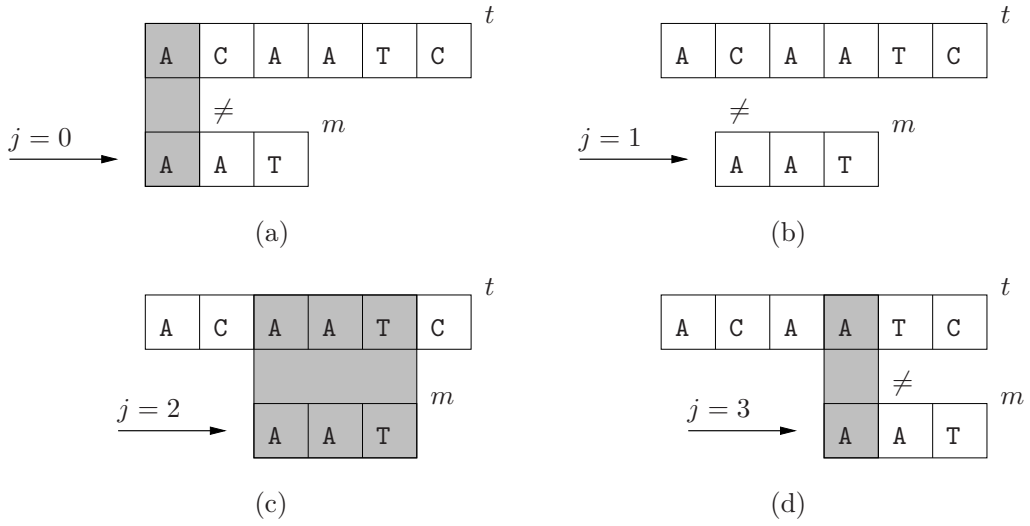

Abbildung 4. Der Sliding-Window-Algorithmus

Dies kann man sich auch so vorstellen, als wäre der ganze Text bis auf ein Fenster in Muster-Breite verdeckt. Das Muster wird dann jeweils mit dem im Fenster sichtbaren Teil des Texts verglichen. Dieses Fenster wird dann einmal vom Anfang des Texts bis zu seinem Ende durchgeschoben. Deshalb nennt man dieses Verfahren auch *Sliding-Window-Algorithmus*.

Hierbei müssen wir im schlimmsten Fall ungefähr (Länge Text mal Länge Muster)-mal zwei Buchstaben miteinander vergleichen. In unserer Anwendung mit einem DNA-Fragment der Länge ca. 1 000 und einem Bakteriengenom der Länge ca. 1 Mio. ergibt sich also ein Aufwand von ca. 1 Milliarde Vergleichen für den Vergleich *eines* Fragments mit *einem* Bakteriengenom der Datenbank. Man beachte, dass wir zur Lösung unserer Aufgabe Tausende von Fragmenten gegen Tausende von Datenbankeinträgen vergleichen müssen.

Ein schlimmstes Beispiel für den Sliding-Window-Algorithmus wäre zum Beispiel gegeben durch einen Text der Form `AAAAAAAA...T` und ein Muster der Form `AAAT`. Auch wenn die in der Anwendung vorkommenden Strings nicht ganz so schlimm aussehen, ist der Aufwand bei diesem Verfahren doch recht hoch.

3.5 Verbesserter Lösungsansatz

Um unser Problem effizient lösen zu können, müssen wir also einen besseren algorithmischen Ansatz finden. Die Idee hierfür ist, den Text nur einmal von links nach rechts zu lesen, sich dabei aber zu merken, welches Anfangsstück des Musters dabei gerade gelesen wurde.

Für das Muster `AGAC` gibt es hierbei fünf verschiedene Situationen, die beim Lesen des Texts auftreten können:

- Das bisher gelesene Anfangsstück endet mit `AGAC`: In diesem Fall haben wir das Muster gefunden.
- Das bisher gelesene Anfangsstück endet mit `AGA`: In diesem Fall fehlt nur noch ein `C`, um das Muster zu vervollständigen.
- Das bisher gelesene Anfangsstück endet mit `AG`: In diesem Fall fehlt noch `AC`, um das Muster zu vervollständigen.
- Das bisher gelesene Anfangsstück endet mit `A` (aber nicht mit `AGA`): In diesem Fall fehlt noch `GAC`, um das Muster zu vervollständigen.
- Das bisher gelesene Anfangsstück endet mit gar keinem Anfangsstück des Musters.

Diese Situationen nennen wir *Zustände*. Mit dem Lesen eines weiteren Buchstabens des Texts wird von einem Zustand in einen anderen übergegangen. Die Übergänge für das Beispielmuster `AGAC` sind in Abbildung 5 gezeigt.

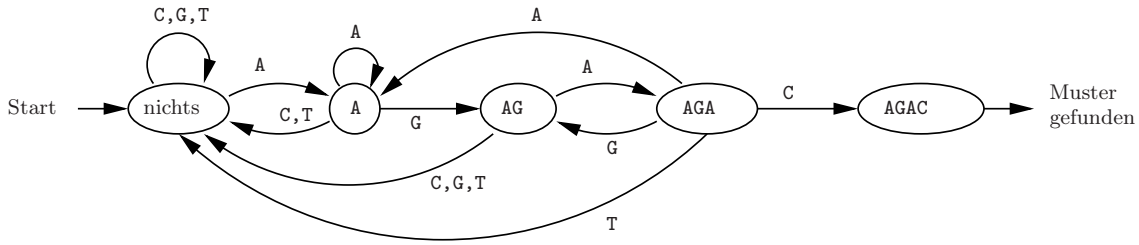

Abbildung 5. Übergangsdiagramm für die Zustände bei der Suche nach dem Muster `AGAC`

Wir bemerken, dass die Zustandsübergänge nur vom Muster, nicht aber vom Text abhängen. Weiterhin lässt sich ein solches Übergangsdiagramm (die Informatiker nennen ein solches Diagramm einen *endlichen Automaten*) effizient erstellen. Damit reduziert sich der Aufwand für die Suche eines Fragments der Länge ca. 1 000 in einem Bakteriengenom der Länge ca. 1 Mio. auf etwa 1 Mio. Übergänge im Diagramm, ist also wesentlich effizienter als der Sliding-Window-Algorithmus.

4 Weiterführende Literatur

[1] Eine kurze Einführung in die biologischen Grundlagen sowie eine ausführliche Darstellung dieser und anderer Methoden zum Vergleich von Strings kann man in dem Buch

> H.-J. Böckenhauer, D. Bongartz: *Algorithmische Grundlagen der Bioinformatik*, Teubner-Verlag 2003

finden. Allerdings richtet sich dies Buch eher an Leser, die schon Grundkenntnisse in der Informatik haben.

[2] Eine gute Einführung in die Grundideen der Informatik, insbesondere auch in die Grenzen algorithmischer Methoden — lesbar für jeden Nicht-Informatiker — findet sich im folgenden Buch:

J. Hromkovič: *Sieben Wunder der Informatik*, Vieweg-Teubner-Verlag 2006.

[3] Eine weitere gute Einführung in das algorithmische Denken, ebenfalls ohne Vorkenntnisse lesbar, gibt das Buch

B. Vöcking, H. Alt, M. Dietzfelbinger, R. Reischuk, C. Scheideler, H. Vollmer, D. Wagner (Hrsg.): *Taschenbuch der Algorithmen*, Springer-Verlag 2008.

[4] N. Silbermann: Warum Zikaden Primzahlen lieben, 2004.
`http://www.welt.de/print-welt/article327049/Warum-Zikaden-Primzahlen-lieben.html`
Zuletzt abgerufen am 23.04.2013.