Grundlegende Algorithmen der Bioinformatik

Vortragsausarbeitungen des gleichnamigen Proseminars

LS 11, Fakultät für Informatik, TU Dortmund

Prof. Dr. Sven Rahmann

mit Beiträgen der TeilnehmerInnen

Wintersemester 2008/09
Entwurf, 15. August 2008

Inhaltsverzeichnis

Vorbemerkungen

Dieses Dokument enthält die Vortragsausarbeitungen des Proseminars Grundlegende Algorithmen der Bioinformatik, das ich im Wintersemester 2008/09 an der Fakultät für Informatik an der TU Dortmund angeobten habe.

Ziel war es, anhand von Bioinformatik-Lehrbüchern grundlegende Algorithmen der Bioinformatik zu erarbeiten, vorzustellen und kurz zusammenzufassen, so dass die wesentlichen Ideen hinter den vorgestellten Verfahren deutlich werden.

Als Veranstalter hoffe ich, dass alle Teilnehmer viel gelernt haben und Spaß bei der Arbeit hatten.

– Prof. Dr. Sven Rahmann, TU Dortmund

KAPITEL 98

Hinweise zur richtigen Benutzung von LaTeX

Ausarbeitung von Sven Rahmann

98.1 Einleitung

In diesem Kapitel stellen wir einige Hinweise zur richtigen Benutzung von LaTeX bereit. Insbesondere werden häufig gemachte Fehler vorgestellt. Zum Beispiel sollte ein Abschnitt (`\section{}`) niemals so kurz sein wie dieser hier.

98.2 Häufig gemachte Fehler

Zu große Abstände nach Satzzeichen. LaTeX setzt hinter einem Punkt einen größeren Abstand als sonst zwischen Wörtern. Normalerweise ist das gewollt, damit zwei Sätze voneinander besser getrennt sind. Es führt aber zu Problemen bei Abkürzungen wie z. B. Prof. Rahmann (beachte die zu großen Abstandslänge; LaTeX versucht schlau zu sein und interpretiert einen einzelnen Buchstaben wie `B.` nicht als Satz). Erstens kann LaTeX einen häufig nicht gewollten Zeilenumbruch einfügen; zweitens sollte der Abstand die normale Länge haben.

Will man einen Zeilenumbruch ausschließen, so kann man einen nichtumbrechenden Zwischenraum (non-breaking space) verweden, in LaTeX geht das mit dem Zeichen `~`. Will man den Umbruch zulassen, muss man dem Leerzeichen einen backslash voranstellen:

```
z.~B.\ Prof.~Rahmann
```

verhindert Umbrüche zwischen `z.` und `B.`, sowie swischen `Prof` und `Rahmann`.

Fehlende Abstände nach Befehlen. Der Befehl `\LaTeX` erzeugt LaTeX. wenn man nun schreibt (wie oben): `\LaTeX setzt`, erhält man: LaTeXsetzt; es fehlt der Abstand! Dasselbe Problem ergibt sich nach allen Befehlen, da das Leerzeichen das Befehlsende markiert.

Zur Lösung kann man entweder den Befehl einklammern, `{\LaTeX} setzt`, oder (einfacher), wieder backslash-space verwenden: `\LaTeX\ setzt`.

Elementare mathematische Funktionen. Variablennamen werden grundsätzlich kursiv geschrieben: i, j, m, n. Dazu wechselt man mit $ in den Mathe-Modus (und auch wieder zurück). Falsch wäre es, hier etwa *italics* text, *i, j, m, n*, zu benutzen (beachte das unterschiedliche Schriftbild).

Konstanten hingegen schreibt man nicht kursiv, insbesondere die imaginäre Einheit $\mathrm{i} = -1$ oder $\mathrm{e} \approx 2.71....$ Im Mathemodus bekommt man das z.B. mit

```
\mbox{\upshape i}
```

hin; eleganter aber ist es, wenn man sich einen eigenen Befehl dafür definiert, zum Beispiel `\imunit` für imaginary unit:

```
\newcommand{\imunit}{\mbox{\upshape i}}
```

Wichtig ist auch, dass Namen elementarer Funktionen wie sin, cos, log nicht kursiv geschrieben werden, ebenso wie Operatorennamen. Dafür stellt LaTeX bereits häufig vordefinierte Befehle zur Verfügung, etwa

`\sin`, `\cos`, `\log`.

Hingegen würde ein vorgebildeter Leser *log* als das Produkt $l \cdot o \cdot g$ interpretieren. Auch das Differential d wird nicht kursiv geschrieben: $\int x \,\mathrm{d}x = x^2/2$. Das wird leider häufig auch in ansonsten guten Büchern falsch gemacht ($\int xdx = x^2/2$ sieht furchtbar aus – beachten Sie die Unterschiede im source code!).

WYSIWYG. Häufig wird der (oft nicht sichtbare) Fehler gemacht, bestimmte Textstellen nur optisch statt logisch auszuzeichnen. Nehmen wir an, dass wir sowohl zu definierende Begriffe als auch Spezies-Namen kursiv hervorheben wollen:

Eine *kontextfreie Grammatik* ist ein 4-Tupel ...
Das Bodenbakterium *C. glutamicum* lebt ...

Wenn wir dies jeweils mit `\textit{}` machen und uns dann entscheiden, dass wir alle zu definierenden Begriffe doch lieber unterstrichen fett haben wollen, bekommen wir ein Problem! Wir müssen jedes `\textit` durchgehen und prüfen, ob es für eine Definition oder anderweitig verwendet wird. Besser ist es, wenn wir eigene Befehle definieren, etwa:

```
\newcommand{\df}[1]{\emph{#1}}
\newcommand{\species}[1]{\textit{#1}}
```

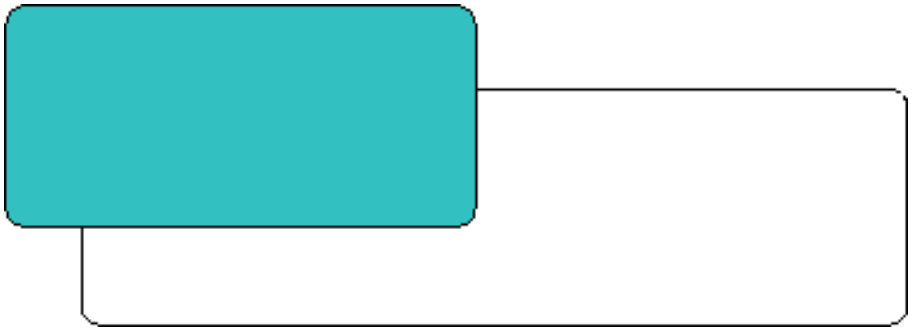

Abbildung 98.1: Ein Kasten

Einbindung von Abbildungen und Tafeln. Keinesfalls sollten Abbildungen und Tafeln direkt in den Text geschrieben werden, da dies im Zweifelsfall zu einem sehr schlechten Seitenumbruch führen kann. Sinnvoller ist es, wenn solche Objekte als frei verschiebbar definiert werden und man Hinweise zur gewünschten Positionierung gibt. Dies geschieht mit Hilfe der `figure` und `table`-Umgebungen.

Wir beschreiben den typischen Fall, das eine Grafik aus einer Datei eingebunden werden soll. Je nachdem, ob mit `latex` nach dvi und dann ps oder mit `pdflatex` direkt in ein pdf übersetzt wird, müssen die Abbildungen als encapsulated postscript (eps) oder als pdf/jpg/png vorliegen. Wenn man beim Dateinamen keine Endung angibt und beide Versionen zur Verfügung stellt, wird die richtige automatisch gewählt.

Der Code zur Einbindung von Abbildung 98.1 sieht so aus:

```
\begin{figure}[t!]
\includegraphics{kasten}
\caption{\label{fig:kasten}Ein Kasten}
\end{figure}
```

Dabei nehmen wir an, dass Dateien `kasten.eps` und `kasten.png` im aktuellen Verzeichnis existieren.

Kleine Tabellen kann man direkt in den Text einbinden: $\begin{array}{|c|r|}\hline x & 1.20 \\ \hline y & 12.30 \\ \hline \end{array}$. Das ist aber nicht so schön. Man könnte sie zwischen zwei Absätze in eine `center`-Umgebung einfügen, allerdings ergibt sich bei größeren Tabellen wieder das Problem des Seitenumbruchs.

x	1.20
y	12.30

Besser setzt man auch Tabellen beweglich in eine `\table`-Umgebung, etwa mit folgendem Code für Tabelle 98.1:

```
\begin{table}[b!]\centering
\begin{tabular}{l|r}
$x$ &  1.20\\\hline
```

```
$y$ & 12.30\\
\end{tabular}
\caption{\label{tab:daten}Wichtige Daten}
\end{table}
```

Wichtig ist auch, dass jedes `figure` oder `table`-Objekt im Text referenziert werden muss, und wenn es durch ein einfaches (siehe Abbildung 98.1) ist. Der Leser will schließlich wissen, *wann* er auf die Abbildung schauen soll.

Jedes Objekt bekommt mit Hilfe von `\label{}` einen Namen, auf den man sich mit `\ref{}` beziehen kann. Auch die Seite, auf der ein Objekt steht, kann man mit `\pageref{}` ausgeben lassen: `Tabelle~\ref{tab:daten} auf Seite~\pageref{tab:daten}` erzeugt: Tabelle 98.1 auf Seite 4.

x	1.20
y	12.30

Tabelle 98.1: Wichtige Daten

KAPITEL 99

Beispielkapitel: Anzahl der 1-Bits in einem Integer

Ausarbeitung von Paul Ruppersberger, SoSe 2008

99.1 Einleitung

Dieses Kapitel widmet sich der Bestimmung der auf 1 gesetzten Bits in einem Integer, dem so genannten „Population Count“ (Im folgenden nur als PopCount bezeichnet). Es wird insbesonders auf Algorithmen zur Berechnung des PopCount und zur Auswertung des PopCount eingegangen. Dazu wird zuerst ein naiver Algorithmus vorgestellt, der im weiteren Verlauf weiter optimiert wird. Grundlegend wird zunächst geklärt warum ein effizienter Algorithmus zur Berechnung des PopCounts überhaupt gebraucht wird.

Anwendungen Der PopCount wird vor allem in der Kodierungs- und Informationstheorie benötigt. Hier hauptsächlich zur Feststellung und Korrektur von Bitfehlern. Dies geschieht anhand des Hamming-Abstands zweier Wörter (Anzahl der Stellen an denen die Wörter unterschiedlich sind). Der Hamming-Abstand dann entspricht der Anzahl der 1-Bits des Terms $x \oplus y$. Aufgrund des engen Zusammenhangs zwischen PopCount und Hamming-Abstand wird der PopCount oft auch als Hamming-Gewicht bezeichnet (Es wird in diesem Kapitel nicht weiter auf den Hamming-Abstand eingegangen). Weitere Anwendungen des PopCounts liegen in der Kryptographie. Hier wird er insbesonders zur Erzeugung von Pseudozufallszahlen verwendet, da es sich um eine Einwegfunktion handelt. D.h. eine Funktion die keine eindeutige Umkehrfunktion hat. Dies ist beim PopCount offensichtlich, da $\text{popcount}\,(1) = \text{popcount}\,(2) = 1$ ist.

Definitionen In diesem Kapitel ist mit einem Integer ein vorzeichenloser 32-Bit-Integer gemeint. Wenn bei der Analye der Algorithmen die Anzahl der Instruktionen angegeben ist, werden Zuweisungen nicht betrachtet.

99.2 Erster Ansatz

Der erste naive Algorithmus den PopCount zu Bestimmen betrachtet alle Bits des Integer und erhöht einen Zähler falls das betrachtete Bit auf 1 gesetzt ist. Die C++ Implementierung ist in Listing 99.1 dargestellt.

Listing 99.1: Implementierung des naiven Algorithmus

```
1 int popcount_1(unsigned int x) {
2     int count = 0;
3     for (int i = 0; i < 32; i++) {
4         if (x & 1) count++;
5         x >>= 1;
6     }
7     return count;
8 }
```

Diese Implementierung enthält sieben Instruktionen innerhalb der For-Schleife, davon zwei Sprünge. Zur Analyse werden folgende Fälle betrachtet:

$$\begin{aligned} \text{Case}_{best} &= 2 + 32 \cdot 6 = 194 \text{ Instruktionen} \\ \text{Case}_{worst} &= 2 + 32 \cdot 7 = 226 \text{ Instruktionen} \\ \text{Case}_{avg} &= 2 + 32 \cdot 6.5 = 210 \text{ Instruktionen} \end{aligned}$$

Im besten Fall besteht der Integer aus 32 0-Bits, dann wird die Anweisung für das Inkrementieren des Zählers ausgelassen. Im schlechtesten Fall sind es 32 1-Bits und im durchschnittlichen Fall enthält der Integer 16 1-Bits. Dieser Algorithmus ist sicherlich nicht sehr elegant, so kann nach der Betrachtung des höchstwertigsten 1-Bits die Schleife abgebrochen werden. Der Ausdruck $x \wedge 1$ liefert 1, wenn das Bit an der niederwertigsten Stelle 1 ist, ansonsten 0. Dies kann verwendet werden um die if-Anweisung innerhalb der Schleife zu eliminieren. Aus diesen Optimierungsansätzen ergibt sich die in Listing 99.2 gezeigte Implementierung.

Listing 99.2: Implementierung des verbesserten naiven Algorithmus

```
1 int popcount_2(unsigned int x) {
2     int count = 0;
3     while (x) {
4         count += (x & 1);
5         x >>= 1;
6     }
7     return count;
8 }
```

Die Schleife enthält jetzt nur noch 5 Instruktionen, davon einen Sprung. Der beste Fall ist trivial bei $x = 0$. Es ist nur die Überprüfung und der Sprung ans Ende der Schleife auszuführen. Der schlechteste Fall benötigt nur noch $\text{Case}_{worst} = 2 + 32 \cdot 5 = 162$ Instruktionen. Einen durchschnittlichen Fall anzugeben fällt schwer, da nicht nur die Anzahl der 1-Bits, sondern auch ihre Position innerhalb des Integers von Bedeutung ist. Daher wird dieser Fall hier nicht betrachtet, da jetzt eine effizientere und elegantere Implementierung betrachtet wird.

Der Ausdruck $x \wedge (x-1)$ liefert x mit dem niederwertigsten 1-Bit auf 0 gesetzt. Dies wird am folgenden Beispiel verdeutlicht:

$$\begin{aligned} x &= 00100110 \\ x-1 &= 00100101 \\ x \wedge (x-1) &= 00100100 \end{aligned}$$

Dieser Ausdruck führt dazu, dass die Schleife nur noch so oft durchlaufen wird bis alle 1-Bits in x eliminiert sind. Die Implementierung ist Listing 99.3 zu entnehmen.

Listing 99.3: Implementierung des optimierten Algorithmus

```
1 int popcount_3(unsigned int x) {
2     int count = 0;
3     while (x) {
4         count++;
5         x &= (x - 1);
6     }
7     return count;
8 }
```

Der beste und schlechteste Fall bleiben dabei gleich. Hier kann auch wieder ein durchschnittlicher Fall angegeben werden, der wie bei der ersten Version des Algorithmus bei 16 1-Bits eintritt; $\text{Case}_{avg} = 2 + 16 \cdot 5 = 82$ Instruktionen. Daraus ergibt sich die Laufzeitfunktion

$$f(n) = 2 + 5n \tag{99.1}$$

wobei n der Anzahl der 1-Bits entspricht.

99.3 Divide-And-Conquer Ansatz

Wie in vielen Bereichen der Algorithmik kann auch in diesem Fall zur Lösung des Problems über eine Lösung nach dem Divide-And-Conquer Prinzip nachgedacht werden. Dazu sind einige Vorüberlegungen notwendig.

Angenommen der PopCount soll für ein 32-Bit Integer berechnet werden und es existiert ein Algorithmus der den PopCount für einen 16-Bit Integer berechnen kann. Dann kann der PopCount für den 32-Bit Integer durch Anwenung dieses Algorithmus auf die linken und die rechten 16-Bit des Integers (Divide-Schritt) und anschließendes Addieren der Ergebnisse (Conquer-Schritt) berechnet werden. Dieses Vorgehen liefert aber leider bei sequentieller Ausführung auf beiden Hälften keinen Geschwindigkeitsvorteil. Hier ist die Zeit, die zur Durchführung der Berechnung benötigt wird, proportional zur Breite des Integers (Bei einem 32-Bit Integer also $32k$ mit k als Proportionalitätskonstante). Bei Aufteilung in zwei 16-Bit breite Teile und anschlieender Addition ergibt sich daraus:

$$16k + 16k + 1 = 32k + 1 \tag{99.2}$$

Einen Zeitvorteil wird erst dann erzielt, wenn es möglich ist die Berechnung der beiden Hälften parallel durchzuführen. Dadurch wird der Aufwand von $32k$ auf $16k + 1$ reduziert. Durch weitere Aufteilung in 8-, 4-, 2- und 1-Bit große Teile wird der triviale Fall bei einer Breite von ein Bit erreicht. Hier ist der Wert des Bits auch der PopCount. Das Prinzip wird duch Abbildung 99.1 veranschaulicht.

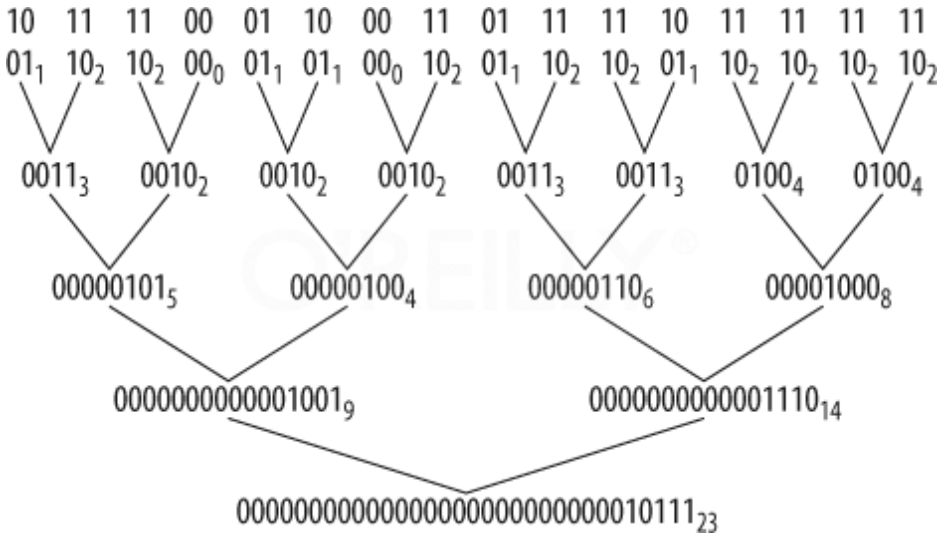

Abbildung 99.1: Divide-And-Conquer Ansatz (Wilson and Oram, 2007, Kapitel 10.2)

Die erste Zeile der Abbildung 99.1 enthält den Integer selbst. In der zweiten Zeile ist dann die erste Stufe des Algorithmus ausgeführt worden. Es werden jeweils die Anzahl der 1-Bits von jeweils zwei Bit in diese zwei Bit geschrieben. Bei der nächsten Stufe werden dann zwei dieser Zweiergruppen zusammengefasst, so dass jeweils Gruppen von vier Bit die Anzahl der 1-Bits in diesen vier Bit enthalten. Dieses wird dann fortgesetzt bis nur noch ein Teil der Breite 32 existiert der das Endergebnis enthält. Eine erste einfache Implementierung sieht wie in Listing 99.4 dargestellt aus.

Listing 99.4: Divide-And-Conquer Algorithmus (frei nach Wikipedia)

```
1  const unsigned int m1  = 0x55555555;
2  const unsigned int m2  = 0x33333333;
3  const unsigned int m4  = 0x0F0F0F0F;
4  const unsigned int m8  = 0x00FF00FF;
5  const unsigned int m16 = 0x0000FFFF;
6
7  int popcount_4(unsigned int x) {
8      x = (x & m1 ) + ((x >>  1) & m1 );
9      x = (x & m2 ) + ((x >>  2) & m2 );
10     x = (x & m4 ) + ((x >>  4) & m4 );
11     x = (x & m8 ) + ((x >>  8) & m8 );
12     x = (x & m16) + ((x >> 16) & m16);
13     return x;
14 }
```

Da dieser Algorithmus keine Schleife benötigt wird eine konstante Laufzeit erreicht. Es werden insgesamt nur 20 Instruktionen für die Berechnung des PopCount benötigt. Allerdings

gibt es auch hier noch Optimierungspotenzial. Bei näherer Betrachtung fällt auf, dass der Ausdruck $x >> 16$ mit 16 0-Bits beginnt und somit die AND-Operation mit der Maske 0x0000FFFF keine Auswirkung mehr hat. Ebenso können AND-Operationen eingespart werden bei denen kein Überlauf bei der Zusammenfassung von zwei Bit-Gruppen auftreten. Die Implementierung dieses Algorithmus ist Listing 99.5 zu entnehmen.

Listing 99.5: Optimierung des Divide-And-Conquer Algorithmus (Wilson and Oram (2007))

```
1  const unsigned int m1 = 0x55555555;
2  const unsigned int m2 = 0x33333333;
3  const unsigned int m4 = 0x0F0F0F0F;
4
5  int popcount_4(unsigned int x) {
6      x = x - ((x >> 1) & m1;
7      x = (x & m2) + ((x >>  2) & m2);
8      x = (x + (x >> 4)) & m4;
9      x = x + (x >> 8 );
10     x = x + (x >> 16);
11     return x & 0x3F; // 00111111
12 }
```

Die Zeile 6 der Funktion aus Listing 99.5 basiert auf den ersten beiden Termen der Gleichung

$$\text{popcount}\,(x) = x - \lfloor x/2 \rfloor - \lfloor x/4 \rfloor - \cdots - \lfloor x/2^{31} \rfloor \tag{99.3}$$

und schreibt die Anzahl der 1-Bits die in diesen zwei Bit enthalten sind in diese zwei Bit (Vgl. Abbildung 99.1). Der Beweis dieser Gleichung wird an dieser Stelle nicht geführt. Diese Implementierung erreicht die konstante Laufzeitfunktion von $f(n) = 15$.

99.4 Rechnen mit PopCounts

Summenbildung Um die Summe von zwei PopCounts zu bilden ist der erste ersichtliche Ansatz sicherlich $\text{popcount}\,(x) + popcount(y)$. Es werden dazu $2 * 15 + 1$ Instruktionen benötigt. Es gibt jedoch eine noch effizientere Methode. Diese verwendet den bereits erläuterten Divide-And-Conquer Algorithmus. Bei Betrachtung der Stufe, in der jeweils die Anzahl der 1-Bits in Gruppen aus vier Bit gebildet werden, fällt auf, dass in diesen vier Bit maximal die Zahl 4 hineingeschrieben wird (Maximaler PopCount auf dieser Stufe). Vier Bit bieten jedoch Platz für einen maximalen PopCount von 15. Es wird also mit x und y bis zur zweiten Stufe getrennt verfahren und anschließend werden beide Zahlen addiert. Das Ergebnis der Addition enthält dann die Anzahl der 1-Bits, die x und y (zusammen) in je vier Bit haben. Jetzt wird genau wie bei der normalen PopCount-Berechnung verfahren. Die Implementierung ist Listing 99.6 zu entnehmen. Es werden statt 31 nur noch 24 Instruktionen benötigt. Dieser Algorithmus funktioniert auch problemlos zur Addition von drei Integern.

Differenzbildung Der Ansatz bei der Subtraktion zweier PopCount ist ähnlich der bei der Addition. Es wird zunächst getrennt von einander der PopCount von x und y berechnet

Listing 99.6: Implementierung des Additionsalgorithmus

```
int popadd(unsigned int x, unsigned int y) {
    x = x - ((x >> 1) & m1;
    x = (x & m2) + ((x >>  2) & m2);
    y = y - ((y >> 1) & m1;
    y = (y & m2) + ((y >>  2) & m2);
    x = x + y;
    x = (x & m4) + ((x >>  4) & m4);
    x = x + (x >> 8 );
    x = x + (x >> 16);
    return x & 0x3F; // 00111111
}
```

und Anschließend werden die Ergebnisse subtrahiert. Aber auch hier kann eleganter verfahren werden. Dazu wird ausgenutzt, dass der PopCount eines Integers gleich 32 minus dem Einerkomplement des Integers ist.

$$\text{popcount}\,(x) = 32 - \text{popcount}\,(\bar{x}) \tag{99.4}$$

Daraus leitet sich dann für die Substraktion folgende Formel ab:

$$\begin{aligned} \text{popcount}\,(x) - \text{popcount}\,(y) &= \text{popcount}\,(x) - (32 - \text{popcount}\,(\bar{y})) \\ \text{popcount}\,(x) - \text{popcount}\,(y) &= \text{popcount}\,(x) + \text{popcount}\,(\bar{y}) - 32 \end{aligned}$$

Es kann für die Substraktion jetzt der gleiche Grundalgorithmus wie für die Addition verwendet werden. Es sind nur zwei weitere Instruktionen notwending, eine für die Invertierung des Integers und eine Subtraktion.

Vergleich zweier PopCounts Um die PopCounts zweier Integer zu vergleichen, also herauszufinden welcher Integer mehr 1-Bits enthält, gibt es zwei Methoden. Bei der Ersten wird einfach die Differenz gebildet und dann das Ergebnis untersucht. Bei der zweiten Methode werden zuerst 1-Bits, die in beiden Integers gesetzt sind, auf 0 gesetzt. Anschließend werden beide Integer solange um je ein 1-Bit reduziert bis einer den Wert 0 hat. Der Integer der zuerst keine 1-Bits mehr enthlt ist dann der Integer mit dem kleineren PopCount. Die Implementierung zeigt Listing 99.7.

Listing 99.7: Implementierung des Vergleichalgorithmus (Wilson and Oram (2007))

```
int popcmp(unsigned int ax, unsigned int ay) {
    unsigned int x = ax & ~ay;
    unsigned int y = ay & ~ax;
    while(true) {
        if (x == 0) return y | -y;
        if (y == 0) return 1;
        x = x & (x-1);
        y = y & (y-1);
    }
}
```

99.5 PopCount eines Arrays

Um den PopCount eines Arrays von Integern zu berechnen wird zunächst die Möglichkeit in Betracht gezogen den PopCount jedes Array-Elements zu bestimmen und die Einzelergebnisse dann zu addieren. Doch wie bereits bei der Addition gesehen existiert ein Algorithmus, der die PopCount-Summe von drei Integern effizient berechnet. Dieser kann auch zur Berechnung des Array-PopCounts herangezogen werden. Dazu wird dieser mit drei Array-Elementen aufgerufen und das Ergebnis zum Gesamtergebnis hinzuaddiert. Bei Array-Größen die sich nicht ohne Rest durch drei teilen lassen, müssen die PopCounts der verbliebenden Elemente mit der herkömmlichen Weise hinzuaddiert werden. Listing 99.8 zeigt wie eine Implementierung ausehen könnte.

Listing 99.8: Implementierung des Array-Algorithmus

```
1  int poparray(unsigned int a[], int size) {
2      int sum = 0;
3      int i;
4
5      for (i = 0; i < size-2; i += 3) {
6          sum += popadd(a[i], a[i+1], a[i+2]);
7      }
8
9      for (; i < size; i++) {
10         sum += popcount(a[i]);
11     }
12
13     return sum;
14 }
```

Literaturverzeichnis

Wikipedia. Hamming weight. Wikipedia, The Free Encyclopedia, 2008. URL `http://en.wikipedia.org/w/index.php?title=Hamming_weight&oldid=215030740`. [Online; Zugriff am 26.05.2008].

G. Wilson and A. Oram, editors. *Beautiful Code*. O'Reilly, 2007.