

Laufzeitanalyse von Algorithmen



6. Mai 2009

Zusammenfassung

Dieses Dokument gibt einen kurzen Überblick über die Laufzeitanalyse (insbesondere die asymptotische Laufzeitanalyse) von Algorithmen. Es wird zunächst die häufig verwendete Landaunotation vorgestellt. Dann wird die Grundlage der Analyse von iterativen und rekursiven Algorithmen behandelt. Anschließend wird auch kurz auf die amortisierte Analyse eingegangen.

Ich übernehme keinerlei Garantie für die Richtigkeit oder Vollständigkeit der Informationen. Dieses Dokument ist nicht als Skriptersatz oder ähnliches gedacht, sondern höchstens als Zusatzlektüre.

Algorithmen, Laufzeit, Analyse, amortisierte Analyse, Landaunotation, O-Notation, Potentialfunktion, Potentialmethode

1 Motivation

Wird ein Verfahren zur Lösung eines Problems entwickelt, so ist man (meistens) “im gleichen Atemzug” daran interessiert, wie schnell das Verfahren läuft. Dazu stellt die Laufzeitanalyse ein solides Mittel dar.

Teil I

Mathematische Grundlagen

2 Konventionen

Einige Konventionen werden bei der Analyse von Algorithmen oft getroffen. Da man das Laufzeit- bzw. Speicherverhalten von Algorithmen untersuchen und mit Funktionen $f(n), g(n)$ beschreiben will (n steht dabei für die Problemgröße), geht man stets von

- positiven
- monoton steigenden

Funktionen aus. Dass eine Laufzeit und der Speicherverbrauch stets positiv sind, ist klar. Auch die andere Bedingung ist einleuchtend, wenn man bedenkt, dass sich (in aller Regel) eine größere Problemgröße auch in einer längeren Laufzeit bzw. in einem höheren Speicherverbrauch niederschlägt.

Daher gehen wir im folgenden stets davon aus, dass wir es mit positiven, monoton steigenden Funktionen zu tun haben.

3 Landaunotation

Die Landaunotation stellt ein relativ flexibles Werkzeug zur Beurteilung von Funktionen bzw. deren Wachstum dar. Sie trifft Aussagen über das Wachstum für hinreichend große Argumente und zeigt an, ob eine Funktion (streng) nach oben und/oder (streng) unten durch eine andere Funktion beschränkt ist.

Ich möchte beispielhaft vorführen, was das Zeichen $O(g(n))$ bedeutet, da dieses am häufigsten verwendet wird. $O(g(n))$ ist eine Menge von Funktionen, die all diejenigen Funktionen beinhaltet, die für $n \rightarrow \infty$ asymptotisch höchstens so schnell wachsen wie $g(n)$. Also gilt:

$$O(g(n)) = \{f(n) \mid \exists c > 0. \exists n_0 \in \mathbb{N}. \forall n \geq n_0. f(n) \leq c \cdot g(n)\}$$

Insgesamt gibt es fünf verschiedene Landausymbole, die im folgenden vorgestellt werden. Es wird gezeigt, wie die Funktion $f(n)$ von $g(n)$ abgeschätzt werden kann.

- Abschätzung nach oben. Die Menge aller Funktionen, die asymptotisch höchstens so schnell wachsen als $g(n)$ wird bezeichnet mit

$$O(g(n)) = \{f(n) \mid \exists c > 0. \exists n_0 \in \mathbb{N}. \forall n \geq n_0. f(n) \leq c \cdot g(n)\}.$$

- Abschätzung nach unten. Die Menge aller Funktionen, die asymptotisch mindestens so schnell wachsen als $g(n)$, wird bezeichnet mit

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0. \exists n_0 \in \mathbb{N}. \forall n \geq n_0. f(n) \geq c \cdot g(n)\}.$$

- Strikte Abschätzung nach oben. Die Menge aller Funktionen, die asymptotisch echt langsamer wachsen als $g(n)$ wird bezeichnet mit

$$o(g(n)) = \{f(n) \mid \forall c > 0. \exists n_0 \in \mathbb{N}. \forall n \geq n_0. f(n) < c \cdot g(n)\}.$$

- Strikte Abschätzung nach unten. Die Menge aller Funktionen, die asymptotisch echt schneller wachsen als $g(n)$ wird bezeichnet mit

$$\omega(g(n)) = \{f(n) \mid \forall c > 0. \exists n_0 \in \mathbb{N}. \forall n \geq n_0. f(n) > c \cdot g(n)\}.$$

- Alle Funktionen, die asymptotisch genau so schnell wie $g(n)$ wachsen, sind in folgender Menge zusammengefasst:

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

Anmerkung: Da wir uns auf positive, monoton wachsende Funktionen beschränkt haben, sind die obigen Definitionen korrekt. Wollte man auch nichtpositive Funktionen mit einbeziehen, so müsste man die Definitionen mit Betragsstrichen versehen:

$$O(g(n)) = \{f(n) \mid \exists c > 0. \exists n_0 \in \mathbb{N}. \forall n \geq n_0. |f(n)| \leq c \cdot |g(n)|\}$$

Oft schreibt man salopp $f(n) = O(g(n))$ für $f(n) \in O(g(n))$. Diese Schreibweise hat sich mittlerweile eingebürgert, führt aber oft genug zu Verständnisproblemen. Ich empfehle stark die etwas striktere Schreibweise mit \in statt $=$. Ein Grund dafür ist, dass man auch $O(f(n)) = O(g(n))$ schreibt und das zwei Bedeutungen haben kann: Einmal die echte Gleichheit und einmal eine Teilmengenbeziehung ($O(f(n)) \subset O(g(n))$).

Ebenso werden die Landausymbole als Platzhalter für eine Funktion verwendet, d.h. $f(n) + O(g(n))$ bedeutet, man addiert zu $f(n)$ eine beliebige Funktion $O(g(n))$ und erhält somit wieder eine Menge von Funktionen.

3.1 Andere Charakterisierungen der Landausymbole

Die obige Beschreibung wird meist bei der Algorithmenanalyse unterrichtet. Es gibt darüberhinaus aber auch noch andere Charakterisierungen der Landaumengen.

$$\begin{aligned}f(n) \in O(g(n)) &\Leftrightarrow 0 \leq \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty \\f(n) \in o(g(n)) &\Leftrightarrow 0 = \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} \\f(n) \in \Omega(g(n)) &\Leftrightarrow 0 < \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq \infty \\f(n) \in \omega(g(n)) &\Leftrightarrow \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \\f(n) \in \Theta(g(n)) &\Leftrightarrow 0 \leq \liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty\end{aligned}$$

4 Beispiele

Es ist vermutlich klar, dass gilt:

$$f(n) \in O(f(n)),$$

denn sei $c = 1, n_0 = 0$. Dann gilt für alle $n \geq n_0 : f(n) < 1 \cdot f(n) \Rightarrow f(n) \in O(f(n))$.

Genau so gilt: $f(n) \in \Omega(f(n))$ und somit auch $f(n) \in \Theta(f(n))$.

4.1 Beweisstrategie

Will man die Zugehörigkeit einer Funktion zu einer Landaumenge $O(g(n))$ oder $\Omega(g(n))$ zeigen, so erfolgt dies oft, indem man sich ein $c > 0$ wählt, ausgehend davon n_0 bestimmt und dann die Relation zeigt.

Will man natürlich die Zugehörigkeit zu einer strikten Menge (also $o(g(n))$ oder $\omega(g(n))$), so kann man sich natürlich nicht auf ein c festlegen, sondern müsste es für alle c zeigen.

Die andere Methode ist eben, über die Limites zu gehen und die Grenzwerte zu berechnen.

4.2 Beispiele zum Selbstlösen

Zeigen Sie:

- Polynome:

$$f(n) = \sum_{i=0}^r c_i \cdot x^i, (c_i \in \mathbb{R}) \Rightarrow f(n) \in O(x^r)$$

$$f(n) = \sum_{i=0}^r c_i \cdot x^i, (c_i \in \mathbb{R}) \Rightarrow f(n) \in \Omega(x^r)$$

- $\log(n!) \in O(n \log n)$
- Logarithmen zu beliebigen Basen $c_1, c_2 > 0$: $\log_{c_1} n \in O(\log_{c_2} n)$
- $f(n), g(n) \in O(h(n)) \Rightarrow f(n) + g(n) \in O(h(n))$

- $O(f(n)) + O(g(n)) = O(f(n) + g(n))$. *Achtung:* Denken Sie daran, dass Sie hier beide Richtungen beweisen müssen!
- $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$. *Achtung:* Auch hier müssen Sie beide Richtungen beweisen.
- $f'(n) \in O(g'(n)) \Rightarrow f(n) \in O(g(n))$. *Anmerkung:* Diese Regel stellt in manchen Fällen eine sehr bequeme Lösung dar, wenn man die Zugehörigkeit für die Ableitungen beweisen kann.

Anmerkung: Die genannten Regeln gelten auch für Ω statt O . Beweisen Sie sie auch für Ω .

Zusammenfassend kann man sagen, dass die Landaunotation Konstante Faktoren und Summanden einfach “rauswirft” und sich “auf den mächtigsten Term beschränkt”.

4.3 Berühmte Fallen

Eine gern gestellte Prüfungsfrage ist die folgende:

Gilt folgendes: $f(n) \in \Theta(g(n)) \Rightarrow f(n) = c \cdot g(n)$, für ein bestimmtes c ?

Diese Aussage gilt *nicht!* Denn: $f(n) \in \Theta(g(n)) \Rightarrow f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$. Hierbei können die Konstanten, die von $O(g(n))$ und $\Omega(g(n))$ “verwendet” werden, durchaus verschieden sein!

Ein weiteres Problem besteht bei Induktion. Wenn die Induktion über eine Addition läuft, wird es falsch, da man bei der Induktion die Addition nicht konstant oft ausführt. Das führt zu Problemen.

Betrachte folgendes: $f(n) = \begin{cases} 3, & \text{falls } n = 0 \\ 3 + f(n-1), & \text{sonst} \end{cases}$. Es ist klar, dass $f(n) = 3 * (n + 1) \in O(n)$.

Nun sehen wir uns an, wie der “Induktionsbeweis” schiefeht. Wir ziehen darauf ab, vermeintlich zu zeigen, dass $f(n) \in O(1)$, also konstant.

Der Induktionsanfang $f(0) = 3 \in O(1)$ ist klar.

Der Induktionsschluss: $f(n+1) = 3 + f(n) \in 3 + O(1) \subset O(1)$. Dies stimmt offensichtlich nicht. Das Problem ist, dass wir hierbei die Addition nicht konstant oft ausgeführt haben.

Daher ein gut gemeinter Rat: Bei Landausymbolen eher Finger weg von Induktion.

Teil II

Grundlegende Analyse von Algorithmen

5 Wie zählt man Operationen?

Eine Frage, die viele Einsteiger vor Probleme stellt, ist die folgende: Wie zähle ich, wie viele Operationen ein Algorithmus macht? Soll ich jede Assemblerinstruktion als Operation werten?

Die zweite Frage ist ganz klar mit “nein” zu beantworten. Ich habe leider nirgends eine vernünftige Erklärung gefunden, wie man die Schritte eines Algorithmus am besten zählt. Daher ein Versuch: Wir zählen die Operationen “nur ganz grob”, nämlich “pro Zeile $O(1)$ ”, sofern keine Schleifen daran teilhaben.

Das heißt, insbesondere rechnen wir für einfache Zuweisungen $O(1)$, also konstanten Aufwand. Ebenso rechnen wir für einfache Bedingungsüberprüfungen auch nur konstanten Aufwand, also wieder $O(1)$.

Für Schleifen rechnen wir $O(f(n))$, wobei $f(n)$ die Anzahl der Durchläufe der Schleife beschreibt.

Außerdem interessiert uns in der grundlegenden Analyse, wie lange ein Algorithmus *im schlechtesten Fall* benötigt (Worst-Case-Laufzeit).

Algorithmus 1 Zahlen 0 bis n addieren.

```
 $i \leftarrow 0$ 
 $sum \leftarrow 0$ 
for  $i \leq n$  do
     $sum \leftarrow sum + i$ 
     $i \leftarrow i + 1$ 
end for
```

5.1 Beispiele

Am besten ist es wahrscheinlich, wenn man ein Beispiel betrachtet:

Algorithmus 1 benötigt für die ersten beiden Zeilen jeweils $O(1)$, da eine einfache Zuweisung eine konstante Zeit benötigt. Also benötigen die ersten beiden Zeilen insgesamt $2 \cdot O(1) = O(1)$.

Anschließend folgt eine Schleife, die genau $n + 1$ mal durchläuft. Es ist bekannt, dass $n + 1 \in O(n)$ liegt. Also benötigt die Schleife an sich $O(n)$ Durchläufe. Der Schleifenkörper besteht nur aus einer Anweisung, kann pro Durchlauf in $O(1)$ ausgeführt werden. Somit benötigt die Schleife insgesamt $O(n) \cdot O(1) = O(n)$ Durchläufe. Der gesamte Algorithmus läuft damit in $O(1) + O(n) = O(n)$.

Algorithmus 2 Zahlen 0 bis n verschachtelter addieren.

```
 $i \leftarrow 0$ 
 $sum \leftarrow 0$ 
for  $i \leq 3 \cdot n$  do
    for  $j \leq 15 \cdot n$  do
         $sum \leftarrow sum + i$ 
         $j \leftarrow j + 1$ 
    end for
     $i \leftarrow i + 1$ 
end for
```

Algorithmus 2 vollbringt ähnliches wie Algorithmus 1. Der einzige Unterschied besteht darin, dass der Schleifenrumpf eine weitere Schleife enthält.

Die äußere Schleife benötigt $O(n)$ Durchläufe (Konstante fällt einfach weg). Die innere Schleife benötigt ebenfalls $O(n)$ Durchläufe (Konstante fällt einfach weg). Die innere Zuweisung läuft natürlich wieder in $O(1)$. Also benötigt die Schleife (und somit der gesamte Algorithmus) insgesamt $O(n) \cdot O(n) \cdot O(1) = O(n^2)$.

Mit ein wenig Übung kann man so sehr bequem die Laufzeit von Algorithmen abschätzen. "Mit der Zeit betrachtet man einfach nur noch die geschachtelten Schleifen, multipliziert deren Maximallängen miteinander, und erhält eine Abschätzung für die Laufzeit."

Algorithmus 3 Bubblesort.

▷ $lst[0 \dots n]$ ist die zu sortierende Liste

```
 $i \leftarrow 0$ 
for  $i \leq n - 1$  do
    for  $j \leq i - 1$  do
        if  $lst[j] > lst[j + 1]$  then
             $swap(lst[j], lst[j + 1])$ 
        end if
         $j \leftarrow j + 1$ 
    end for
     $i \leftarrow i + 1$ 
end for
```

Algorithmus 3 hat wieder zwei Schleifen. Die äußere Schleife benötigt $O(n)$ Durchläufe. Die innere Schleife benötigt maximal $O(n)$ Durchläufe. Somit hätten wir eine Laufzeit von $O(n^2)$.

Tatsächlich jedoch nur $O(i)$ Durchläufe in Abhängigkeit von i . Mit ein wenig Nachdenken sieht man, dass der innere Schleifenrumpf $(\sum_{i=0}^n i) = \frac{n \cdot (n+1)}{2}$ mal aufgerufen wird. Insgesamt erhalten wir also (immer noch) eine Laufzeit von $O(n^2)$.

5.2 Eine Tücke der Landaunotation

Da die Landaunotation konstante Faktoren “absorbiert”, kann es sein, dass Algorithmen mit $O(n)$ in der Praxis eine schlechtere Laufzeit haben als beispielsweise Algorithmen mit Laufzeit $O(n \log n)$. Ist nämlich das zu wählende c für $O(n)$ so groß, dass es $\log n$ bei weitem übersteigt, so kann es sich durchaus lohnen, auf den asymptotisch zwar schlechteren, für genügend kleine n jedoch besseren Algorithmus zurückzugreifen. Die Landaunotation gibt – wie gesagt – nur Aufschluss über das *asymptotische* Laufzeitverhalten.

6 Rekursive Algorithmen

Wie wir iterative Algorithmen analysieren, haben wir bereits gesehen. Rekursive Algorithmen sind meist jedoch etwas komplizierter in der Handhabung.

Algorithmus 4 *recurse*(n, a)

Require: $n = 2^k$, für $k \in \mathbb{N}$

```

if  $n = 1$  then
    return  $a$ 
end if
 $result \leftarrow 0$ 
 $result \leftarrow recurse(\frac{n}{2}, 1) + recurse(\frac{n}{2}, -1)$ 
 $i \leftarrow 0$ 
for  $i < n$  do
     $result \leftarrow result + i$ 
     $i \leftarrow i + 1$ 
end for

```

Betrachten wir dazu Algorithmus 4, über dessen Sinnhaftigkeit man zwar streiten kann, der aber anschaulich ist. Wie analysieren wir die Laufzeit $T(n)$ eines solchen Algorithmus am besten?

Stellen wir zunächst fest, dass bis auf die Rekursion und die Schleife alles in $O(1)$ geschieht. Diese Bestandteile lasse ich im folgenden weg, da sie allein schon von der Laufzeit der Schleife $O(n)$ geschluckt werden. Wie lasse ich jedoch die Rekursion in die Berechnung von $T(n)$ einfließen?

Das geschieht folgendermaßen:

$$T(n) = O(n) + 2 \cdot T\left(\frac{n}{2}\right) \leq c \cdot n + 2 \cdot T\left(\frac{n}{2}\right)$$

Nun gibt es drei Möglichkeiten:

- Man ist mathematisch so fit, und schafft es, die Rekursion aufzulösen. Diese Methode ist zwar elegant, jedoch oft zu aufwändig.
- Man setzt ein paar Rekursionen einfach ein, und sieht, welches Muster sich abzeichnet. Das vermutete Muster müsste man dann noch (meist über Induktion¹) beweisen.

¹Wie zuvor erwähnt, sollte diese Induktion ohne Landausymbole von statten gehen.

- Man erledigt es mit dem Master-Theorem, das viele solcher algorithmisch interessanter Gleichungen abdeckt. Es besagt folgendes:

Sei

$$T(n) = \begin{cases} a, & \text{falls } n = 1 \\ c \cdot n + d \cdot T\left(\frac{n}{b}\right), & \text{sonst} \end{cases}$$

Dann gilt:

$$T(n) \in \begin{cases} \Theta(n), & \text{falls } d < b \\ \Theta(n \log n), & \text{falls } d = b \\ \Theta(n^{\log_b d}), & \text{falls } d > b \end{cases}$$

Damit können wir Algorithmus 4 fertig analysieren. In diesem Fall ist $d = 2 = b$, also ist $T(n) \in \Theta(n \log n)$ und somit insbesondere $T(n) \in O(n \log n)$.

Teil III

Amortisierte Analyse von Algorithmen

7 Einleitung

Die amortisierte Analyse bietet sich an, wenn man in einer Folge von Operationen (seien es Operationen in einem Algorithmus oder Operationen auf einer Datenstruktur) viele sehr günstige Operationen dabei hat und einige wenige teurere Operationen. Dann scheint es nur logisch, dass man bei der Analyse nicht immer den Worst-Case betrachten sollte, sondern irgendwie über die Operationen mitteln sollte, um eine realistische Abschätzung zu erhalten.

8 Die Potentialmethode

Wir betrachten eine Folge von Operationen (op_i) auf einer Datenstruktur, die Kosten (c_i) mit sich bringt². Operation op_i versetzt die Datenstruktur vom Zustand z_{i-1} nach z_i . Die Zustandsmenge sei bezeichnet mit Z . Sei n die Anzahl der Operationen insgesamt. Dann gilt, dass die Gesamtkosten $c = c_1 + c_2 + \dots + c_{n-1} + c_n$ betragen. Die Kosten pro Operation im Schnitt sind dann $\frac{c}{n}$.

Nun führt man eine Potentialfunktion $\phi : Z \rightarrow \mathbb{N}$ ein, für die gelten soll, dass $\forall z_i \in Z : \phi(z_0) \leq \phi(z_i)$, wobei z_0 den Startzustand der Datenstruktur bezeichne. Nun definiert die *amortisierten Kosten*

$$a_i = c_i + \phi(z_i) - \phi(z_{i-1}).$$

Mit dieser Festlegung gilt dann, dass die Summe der realen Kosten nach oben abschätzbar ist durch die Summe der amortisierten Kosten:

$$\sum_{i=1}^n c_i \leq \sum_{i=1}^n a_i,$$

da sich die rechte Summe zu einer Teleskopsumme ergibt und nur noch die ganzen c_i und $-\phi(z_0) + \phi(z_n) \geq 0$ übrigbleiben. Das heißt, wir haben eine obere Schranke für die gesamten Kosten gewonnen.

²Bei der amortisierten Laufzeitanalyse entfernt man sich meist von der Landaunotation, da sie das ganze nur unnötig kompliziert machen würde. So verwendet man statt $O(n)$ einfach nur noch n

Nun kommt der Trick: Wenn es auch für die amortisierten Kosten eine obere Schranke A gibt, also $a_i \leq A \forall a_i$, dann gilt, dass

$$\sum_{i=1}^n c_i \leq n \cdot A,$$

was bedeutet, dass wir amortisiert (also betrachtet über die ganze Folge von Operationen) für jede Operation höchstens A benötigen haben.

9 Wahl der Potentialfunktion

So weit, so gut. Die Idee der Potentialfunktion ist (meines Erachtens) relativ gut nachvollziehbar. Was jedoch *nicht* von vorn herein klar ist, ist die Frage, wie man die Potentialfunktion wählen soll.

Das kann man nicht pauschal beantworten, jedoch versuche ich ein paar Tipps zu geben, an die man sich halten kann, wenn man auf der Suche nach einer Potentialfunktion ist.

Genauere Analyse der Operationen Überlegen Sie sich gut, wie die Kosten der Operationen ausfallen und überprüfen Sie, wovon die Kosten abhängen. Binden Sie diese Größen in die Potentialfunktion mit ein.

Charakteristische Elemente einbauen Es dürfte wohl klar sein, dass man in die Potentialfunktion die charakteristischen Elemente, die den aktuellen Zustand ausmachen, auf geeignete Weise einbaut. Charakteristische Elemente sind beispielsweise: Auslastung der Datenstruktur, Gesamtgröße der Datenstruktur, durch Operationen veränderbare Größen.

Wann treten welche Operationen auf? Oft ist es bei der amortisierten Analyse so, dass bestimmte Operationen gar nicht zu jedem beliebigen Zeitpunkt auftreten können, sondern nur zu bestimmten. Richten Sie die Potentialfunktion danach aus.

Versuche, auszugleichen Dieser Punkt ist meiner Ansicht nach der wichtigste. Da die amortisierten Kosten als $a_i = c_i + \phi(z_i) - \phi(z_{i-1})$ definiert sind und man die obere Schranke A möglichst klein halten will, muss man sich überlegen, wie man teure Operationen (also große c_i) möglichst weit "herunterdrosselt" und gleichzeitig günstige Operationen (also kleine c_i) nicht allzu stark in die Höhe treibt.

Bei großem c_i muss also $\phi(z_{i-1})$ im Verhältnis zu $\phi(z_i)$ so groß sein, dass der Wert $\phi(z_i) - \phi(z_{i-1})$ "gut nach unten zieht". Also muss man bei teuren Operationen das Potential des Zustands davor recht groß im Verhältnis zum Potential des Zustands danach konstruieren.

10 Beispiel Binärzähler

Betrachten wir das Beispiel eines händisch implementierten Binärzählers, der mit dem Wert 0 startet und in jedem Schritt um 1 hochgezählt wird. Es ist sofort klar, dass nicht jeder Inkrementenschritt die gleiche Anzahl an Bits kippt.

Betrachten wir einen kurzen Ausschnitt:

$$0000 \xrightarrow{1} 0001 \xrightarrow{2} 0010 \xrightarrow{1} 0011 \xrightarrow{3} 0100 \xrightarrow{1} 0101$$

Über den Pfeilen stehen jeweils die Kosten der Inkrementoperation³.

Eine kurze Überlegung bezüglich des Erhöhungsschrittes ergibt, dass von unten (also vom niedrigwertigsten Bit ab) so viele Bits gekippt werden müssen, bis man auf die erste 0 trifft. Es müssen nämlich die Einsen zu Nullen umgewandelt werden und eventuell ein Übertrag mitgeschleppt werden

³Auch hier wurde von der Landaunotation abgewichen und nur noch die Anzahl der gekippten Bits notiert.

bis man zu einer Null kommt, zu der man den Übertrag addieren kann. Dann muss noch die gefundene Null gekippt werden.

Diese Erkenntnis versuchen wir nun in die Potentialfunktion mit einzubeziehen. Wir bezeichnen mit b_i die Anzahl der Bits, die gekippt werden müssen im i -ten Inkrementschritt (b_i entspricht genau c_i , also den Kosten). Es ist leicht zu erkennen, dass $b_i = 1 + \#$ hängende Einsen des Binärzählers (hängende Einsen sind die niederwertigsten Einsen, die bis zur ersten Null kommen).

Nun sehen wir zu, dass wir den Term $c_i + \phi(z_i) - \phi(z_{i-1})$ immer möglichst klein halten. Dazu versuchen wir, ϕ so zu wählen, dass es c_i möglichst gut ausgleicht. Da in einem Inkrementschritt die Anzahl der hängenden Einsen um b_i abnimmt

Betrachten wir dazu zwei Fälle:

- Der Binärzähler hat als niederwertigste Stelle eine 0. Dann ist $c_i = 1$, da nur ein Bit gekippt werden muss. Die Anzahl der hängenden Einsen nimmt in diesem Fall um 1 zu.
- Der Binärzähler hat als niederwertigste Stelle eine 1. Dann ist $c_i = b_i$. Die Anzahl der hängenden Einsen nimmt in diesem Fall um $b_i - 1 = c_i - 1$ ab.

Man sieht sehr schön, dass man mit folgender Definition von ϕ die Schranke gut nach unten bringt:

$$\phi = \# \text{hängende Einsen}$$

Dann ist nämlich $a_i = c_i + \phi(z_i) - \phi(z_{i-1})$ entweder $c_i + 1 = 2$ (falls die niederwertigste Stelle eine Null war) oder $c_i - (b_i - 1) = c_i - c_i + 1 = 1$ (falls die niederwertigste Stelle eine 1 war). Somit erhalten wir als obere Schranke die 2 und können bestätigen, dass amortisiert nur Laufzeitkosten 2 auftreten und somit amortisiert konstante Laufzeit herrscht.

Es wird an diesem Beispiel sehr schön klar, wie man versucht hat, die Kosten “genau umgekehrt” in die Potentialdifferenz zu packen und so die Schranke zu drücken.

Anmerkung: Die Schranke A muss nicht notwendigerweise eine Zahl sein. Es kann sich hierbei auch um $O(\log n)$ oder ähnliches handeln.