technische universität
dortmund

Algorithmen auf Sequenzen

Bitsequenzen

Dominik Kopczynski

Lehrstuhl für Algorithm Engineering (LS11)
Fakultät für Informatik
TU Dortmund

Begriffe und Definitionen

Einige Beispiele für Sequenzen sind:

- Biosequenzen (DNA, RNA, Proteine)
- Texte (Literatur, wissenschaftliche Texte)
- Quelltexte von Programmen
- Dateien, Datenströme
- Zeitreihen, Spektren (Audiosignale, Massenspektren, ...)
- Bilder, Videos

Begriffe und Definitionen

Folgende Probleme umfasst die Sequenzanalyse:

- Mustersuche - exakt oder fehlertolerant
- Sequenzvergleich - Ermitteln von Gemeinsamkeiten und Unterschieden
- Kompression - Erkennung und Ausnutzung von kompressionsfreudigen Sequenzstrukturen
- Muster- und Signalentdeckung - Suche nach unbekannten auffälligen Mustern

Begriffe und Definitionen

Folgende Probleme umfasst die Sequenzanalyse:

- **Mustersuche - exakt oder fehlertolerant**
- **Sequenzvergleich - Ermitteln von Gemeinsamkeiten und Unterschieden**
- Kompression - Erkennung und Ausnutzung von kompressionsfreudigen Sequenzstrukturen
- ~~Muster- und Signalentdeckung - Suche nach unbekannten auffälligen Mustern~~

In diesem Semester

Begriffe und Definitionen

Definitionen:

- Ein *Alphabet* Σ ist eine (endliche oder unendliche) Menge.
- Eine *Indexmenge* $\mathcal{I}$ ist eine endliche oder abzählbar unendliche linear geordnete Menge. Beispiele hierfür sind $\mathbb{N}, \mathbb{Z}$ und die Ordnung $\leq$.
- Eine *Sequenz* ist eine Funktion $s : \mathcal{I} \to \Sigma$ oder äquivalent, ein Tupel $s \in \Sigma^{\mathcal{I}}$.

In der Vorlesung befassen wir uns ausschließlich mit endlichen Sequenzen, also $\mathcal{I} = \{0, \ldots, n-1\}$ (Indizierung beginnt bei 0) für $n \in \mathbb{N}$. Zur Vereinfachung schreiben wir Σ^n.

Begriffe und Definitionen

Beispiele für Sequenzen über verschiedenen Alphabeten:

Sequenztyp	**Alphabet Σ**
DNA-Sequenz	$\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$
Protein-Sequenz	20 Standard-Aminosäuren
C-Programme	ASCII-Zeichen (7-bit)
Java-Programme	Unicode-Zeichen
Audiosignal (16-bit samples)	$\{0, \ldots, 2^{16} - 1\}$
Massenspektrum	Intervall $[0, 1]$ (unendlich) oder Double

Begriffe und Definitionen

Definitionen:

- Die Elemente von Σ^n nennt man *Wörter*, *Tupel*, *Strings*, *Sequenzen* der Länge n, sowie $n-$*Mere* oder $n-$*Gramme*.
- Sei $\Sigma^+ := \bigcup_{n \geq 1} \Sigma^n$ die Menge aller endlichen Strings der Länge $n \geq 1$.
- Sei $\Sigma^* := \bigcup_{n \geq 0} \Sigma^n$ die Menge aller *endlichen* Strings über Σ, wobei $\Sigma^0 = \{\epsilon\}$ und ϵ *der leere String* ist.

Begriffe und Definitionen

Teilstring, Teilsequenz:

- Sei $s \in \Sigma^*$ ein String. Sei $s[i]$ der Buchstabe, der in s an der Stelle i steht. Dabei muss $i \in \mathcal{I}$ sein. Wir schreiben $s[i:j]$ für den *Teilstring* von i bis j (ausschließlich). Falls $i \geq j$, ist per Definition $s[i:j] = \epsilon$.
- Eine *Teilsequenz* von s definieren wir als $(s_i)_{i \in I}$ mit $I \subset \mathcal{I}$.

Eine Teilsequenz ist im Gegensatz zum Teilstring also nicht notwendigerweise zusammenhängend. Die Begriffe Teilstring und Teilsequenz sind daher auseinanderzuhalten.

Begriffe und Definitionen

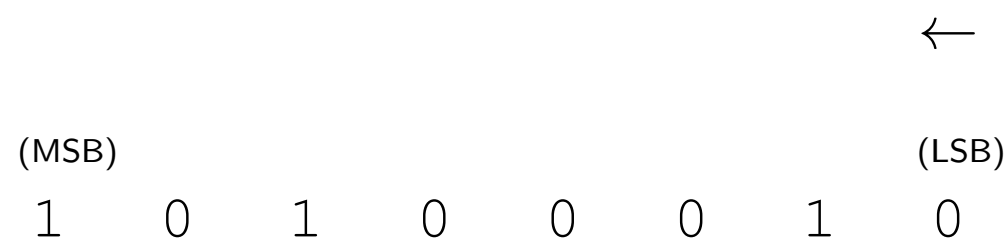
Präfix, Suffix:

- Sei $s[:i] := s[0:i]$ der beginnende Teilstring *(Präfix)* von s.
- Sei $s[i:] := s[i:|s|]$ der endende Teilstring *(Suffix)* von s.

Wenn t ein Präfix (Suffix) von s ist und $t \neq \epsilon$ und $t \neq s$, dann bezeichnen wir t als *echtes Präfix (Suffix)* von s.

Bitsequenzen

- Einfachste Form von Sequenzen.
- Alphabet besteht nur aus zwei Buchstaben, nämlich $\{0, 1\}$ oder $\{\texttt{false}, \texttt{true}\}$.
- Bitsequenzen sind bereits aus Rechnerstrukturen bekannt.
- Die Leserichtung geht vom *least significant bit* (LSB) zum *most significant bit* (MSB).



Darstellung im Rechner

- Abhängig von der Maschinenarchitektur besteht ein Wort aus $W = \{16, 32, 64, 128\}$ Bits.
- Um n Bits zu speichern, braucht man also $\lceil n/W \rceil$ Wörter.
- Verschiedene Bitoperationen sind hardwareseitig implementiert.

Bitoperationen

Verschiedene Operationen sind neben arithmetischen Operationen auf der Bitsequenz s möglich.

Bitweise Negation $\sim s$ ist definiert als $(\sim s)[i] := \sim s[i]$, wobei

$\sim$	0	1
$=$	1	0

Bitweise Verundung $s \,\&\, t$ ist definiert als $(s \,\&\, t)[i] := s[i] \,\&\, t[i]$, wobei

	0	1	0	1
&	0	0	1	1
$=$	0	0	0	1

Bitweise Veroderung $s \mid t$ ist definiert als $(s \mid t)[i] := s[i] \mid t[i]$, wobei

	0	1	0	1
$\mid$	0	0	1	1
$=$	0	1	1	1

Bitoperationen

Weitere Operationen sind:

Bitweise exklusive
Veroderung $s \oplus t$ ist definiert als
$(s \oplus t)[i] := s[i] \oplus t[i]$, wobei

	0	1	0	1
$\oplus$	0	0	1	1
$=$	0	1	1	0

Bitoperationen

Shiftoperation:
Wir betrachten auf den W Bits von $s = (s[k])_{k=0}^{W-1}$ nun die Operationen *Linksverschiebung* $\ll$

$$(s \ll b)[i] := \begin{cases} s[i-b] & \text{wenn } 0 \leq i-b \\ 0 & \text{sonst} \end{cases}$$

und *Rechtsverschiebung* $\gg$

$$(s \gg b)[i] := \begin{cases} s[i+b] & \text{wenn } i+b < W \\ 0 & \text{sonst} \end{cases}$$

um jeweils $b \geq 0$ Bits.

Bitoperationen

Multiplikation und ganzzahlige Division von 2er-Potenzen:

Linksshift um b Bits entspricht Multiplikation um 2^b:

$(s)_{10}$	s	b	$(s \ll b)_{10}$	$s \ll b$
5	101	1	10	1010
12	1100	2	48	110000

Bitoperationen

Multiplikation und ganzzahlige Division von 2er-Potenzen:

Linksshift um b Bits entspricht Multiplikation um 2^b:

$(s)_{10}$	s	b	$(s \ll b)_{10}$	$s \ll b$
5	101	1	10	1010
12	1100	2	48	110000

Rechtsshift um b Bits entspricht ganzzahliger Division um 2^b:

$(s)_{10}$	s	b	$(s \gg b)_{10}$	$s \gg b$
17	10001	1	8	1000
163	10100011	3	20	10100

Bitoperationen

Multiplikation und ganzzahlige Division von 2er-Potenzen:

Linksshift um b Bits entspricht Multiplikation um 2^b:

$(s)_{10}$	s	b	$(s \ll b)_{10}$	$s \ll b$
5	101	1	10	1010
12	1100	2	48	110000

Rechtsshift um b Bits entspricht ganzzahliger Division um 2^b:

$(s)_{10}$	s	b	$(s \gg b)_{10}$	$s \gg b$
17	10001	1	8	1000
163	10100011	3	20	10100

Achtung: Bits können „rausfliegen“

$$\boxed{01001110} \quad \ll 2 \quad \boxed{00111000}$$

oder wenn $b \geq W$ ist das Verhalten in vielen CPUs nicht definiert.

Bitoperationen

Modulorechnung:
Der ganzzahlige Rest einer Division lässt sich sehr einfach auf 2er-Potenzen anwenden. Nur die niederwertigsten b Bits werden beibehalten, alle höherwertigen Bits werden auf 0 gesetzt (siehe Präfix).

$$(s \ \% \ 2^b)[i] := \begin{cases} s[i] & \text{wenn } i < b \\ 0 & \text{sonst} \end{cases}$$

Operation durch Verundung mittels einer Maske m möglich,
Beispiel $01001100 \ \% \ 2^5$:

$$\begin{array}{rll} & s & 01001100 \qquad\qquad\qquad\qquad\qquad\qquad 76_{10} \\ \& & m & \\ \hline & = & \end{array}$$

Bitoperationen

Modulorechnung:
Der ganzzahlige Rest einer Division lässt sich sehr einfach auf 2er-Potenzen anwenden. Nur die niederwertigsten b Bits werden beibehalten, alle höherwertigen Bits werden auf 0 gesetzt (siehe Präfix).

$$(s \ \% \ 2^b)[i] := \begin{cases} s[i] & \text{wenn } i < b \\ 0 & \text{sonst} \end{cases}$$

Operation durch Verundung mittels einer Maske m möglich,
Beispiel $01001100 \ \% \ 2^5$:

$$\begin{array}{rlll} & s & 01001100 & & 76_{10} \\ \& & m & 00000001 \quad \widehat{=} \ 1 & & 1_{10} \\ \hline & = & & & \end{array}$$

Bitoperationen

Modulorechnung:
Der ganzzahlige Rest einer Division lässt sich sehr einfach auf 2er-Potenzen anwenden. Nur die niederwertigsten b Bits werden beibehalten, alle höherwertigen Bits werden auf 0 gesetzt (siehe Präfix).

$$(s \ \% \ 2^b)[i] := \begin{cases} s[i] & \text{wenn } i < b \\ 0 & \text{sonst} \end{cases}$$

Operation durch Verundung mittels einer Maske m möglich, Beispiel $01001100 \ \% \ 2^5$:

$$\begin{array}{rlll} & s & 01001100 & & 76_{10} \\ \& & m & 00100000 & \widehat{=} (1 \ll 5) & 32_{10} \\ \hline & = & & & \end{array}$$

Bitoperationen

Modulorechnung:
Der ganzzahlige Rest einer Division lässt sich sehr einfach auf 2er-Potenzen anwenden. Nur die niederwertigsten b Bits werden beibehalten, alle höherwertigen Bits werden auf 0 gesetzt (siehe Präfix).

$$(s \ \% \ 2^b)[i] := \begin{cases} s[i] & \text{wenn } i < b \\ 0 & \text{sonst} \end{cases}$$

Operation durch Verundung mittels einer Maske m möglich,
Beispiel $01001100 \ \% \ 2^5$:

$$\begin{array}{rlll} & s & 01001100 & & 76_{10} \\ \& & m & 00011111 & \widehat{=} (1 \ll 5) - 1 & 31_{10} \\ \hline & = & & & \end{array}$$

Bitoperationen

Modulorechnung:
Der ganzzahlige Rest einer Division lässt sich sehr einfach auf 2er-Potenzen anwenden. Nur die niederwertigsten b Bits werden beibehalten, alle höherwertigen Bits werden auf 0 gesetzt (siehe Präfix).

$$(s \ \% \ 2^b)[i] := \begin{cases} s[i] & \text{wenn } i < b \\ 0 & \text{sonst} \end{cases}$$

Operation durch Verundung mittels einer Maske m möglich, Beispiel $01001100 \ \% \ 2^5$:

	s	01001100		76_{10}
&	m	00011111	$\widehat{=} (1 \ll 5) - 1$	31_{10}
	$=$	00001100		12_{10}

Bitoperationen

Modulorechnung:
Achtung: Verhalten undefiniert, wenn ganzes Wort maskiert werden soll, da die Operation $1 \ll W$ angewendet wird.

Manipulieren von Bits

Da man nur auf die Worte und nicht auf die Bits im RAM zugreifen kann, muss für das Bit an Index i zuerst das j-te Wort bestimmt werden, in dem es gespeichert ist. Sei dazu

$$B := (B[j])_{j=0}^{\lceil n/W \rceil - 1}, B[j] = s[l : r]$$

mit $l = j \cdot W, r = j \cdot W + W$ die Sequenz der Maschinenwörter. Um j zu ermittlen, wird die ganzzahlige Division genutzt: $j := i \gg w$, wobei $w = \lceil \log W \rceil$.

Wichtig: wenn nicht anders definiert, dann gilt

$$\log x := \log_2 x$$

im weiteren Verlauf aller Folien.

Manipulieren von Bits

Das k-te Bit innerhalb eines Wortes kann gesetzt, gelöscht oder invertiert (toggle) werden.

- Setzen von Bits: $B[j] = B[j] \mid (1 \ll k)$
- Löschen von Bits: $B[j] = B[j] \,\&\, \sim(1 \ll k)$
- Invertieren von Bits: $B[j] = B[j] \oplus (1 \ll k)$

Popcount

Die Funktion *popcount* zählt die vorkommenden 1en innerhalb eines Wortes.

- Auch bekannt als *population count*, *popcnt* oder Hemming-Gewicht.
- Popcount wird in verschiedenen Datenstrukturen und u.a. in der Kryptographie eingesetzt.
- Neuere Rechner, wie Intel SSE4.2, bieten Popcount-Befehl als Maschinencode an.
- C-Befehl lautet:
  `int __builtin_popcount(unsigned int x)`

Popcount

Naive Implementierung, alle Bits aufaddieren:

```
1 int popcount(uint64 x) {
2     int count;
3     for (count=0; x; x>>=1) count += x & 1;
4     return count;
5 }
```

Alternative: Look-Ahead-Tabelle:

```
1 int popcount_la[] = {0, 1, 1, 2, 1, 2, 2, 3, ...};
```

Popcount

Naive Implementierung, alle Bits aufaddieren: Laufzeit $\mathcal{O}(W)$.

```
1 int popcount(uint64 x) {
2     int count;
3     for (count=0; x; x>>=1) count += x & 1;
4     return count;
5 }
```

Alternative: Look-Ahead-Tabelle: Speicherplatz $\mathcal{O}(2^W)$.

```
1 int popcount_la[] = {0, 1, 1, 2, 1, 2, 2, 3, ...};
```

Geht es effizienter?

Popcount

Parallelisierung ausnutzbar, Wort wird in kleine Blöcke aufgeteilt.

- Werte innerhalb der Blöcke werden aufaddiert.
- Masken werden zur Separierung der Blöcke verwendet.
- Größe der Blöcke wird sukzessiv verdoppelt.

Popcount

1) Bits innerhalb 2er Blöcke addieren, dazu 1er-Maske M_1 nutzen.

$$x \quad \overset{15}{0}1011110010011\overset{0}{0}1$$

Popcount

1) Bits innerhalb 2er Blöcke addieren, dazu 1er-Maske M_1 nutzen.

	15															0
x	0	1	0	1	1	1	1	0	0	1	0	0	1	1	0	1
M_1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Popcount

1) Bits innerhalb 2er Blöcke addieren, dazu 1er-Maske M_1 nutzen.

	15															0
x	0	1	0	1	1	1	1	0	0	1	0	0	1	1	0	1
M_1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
x & M_1	0	1	0	1	0	1	0	0	0	1	0	0	0	1	0	1

Popcount

1) Bits innerhalb 2er Blöcke addieren, dazu 1er-Maske M_1 nutzen.

	15															0
x	0	1	0	1	1	1	1	0	0	1	0	0	1	1	0	1
M_1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
x & M_1	0	1	0	1	0	1	0	0	0	1	0	0	0	1	0	1
$+(x \gg 1)$ & M_1	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	0

Popcount

1) Bits innerhalb 2er Blöcke addieren, dazu 1er-Maske M_1 nutzen.

	15															0
x	0	1	0	1	1	1	1	0	0	1	0	0	1	1	0	1
M_1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$x \,\&\, M_1$	0	1	0	1	0	1	0	0	0	1	0	0	0	1	0	1
$+(x \gg 1) \,\&\, M_1$	0	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0
$x =$	0	1	0	1	1	0	0	1	0	1	0	1	0	0	1	1

Popcount

2) Bits innerhalb 4er Blöcke addieren, dazu 2er-Maske M_2 nutzen.

$$x \quad \overset{15}{0101}\,\overset{}{1001}\,\overset{}{0101}\,\overset{0}{1001}$$

Popcount

2) Bits innerhalb 4er Blöcke addieren, dazu 2er-Maske M_2 nutzen.

	15			0
x	0101	1001	0100	1001
M_2	0011	0011	0011	0011

Popcount

2) Bits innerhalb 4er Blöcke addieren, dazu 2er-Maske M_2 nutzen.

	15			0
x	0101	1001	0100	1001
M_2	0011	0011	0011	0011
$x \,\&\, M_2$	0001	0001	0000	0001

Popcount

2) Bits innerhalb 4er Blöcke addieren, dazu 2er-Maske M_2 nutzen.

	15			0
x	0101	1001	0100	1001
M_2	0011	0011	0011	0011
$x \,\&\, M_2$	0001	0001	0000	0001
$+(x \gg 2) \,\&\, M_2$	0001	0010	0000	1010

Popcount

2) Bits innerhalb 4er Blöcke addieren, dazu 2er-Maske M_2 nutzen.

	15			0
x	0101	1001	0100	1001
M_2	0011	0011	0011	0011
$x \,\&\, M_2$	0001	0001	0000	0001
$+(x \gg 2) \,\&\, M_2$	0001	0010	0000	1010
$x =$	0010	0011	0000	1011

Popcount

3) Bits innerhalb 8er Blöcke addieren, dazu 4er-Maske M_3 nutzen.

$$x \quad \overset{15}{0010001100010011}\overset{0}{}$$

Popcount

3) Bits innerhalb 8er Blöcke addieren, dazu 4er-Maske M_3 nutzen.

	15	0
x	0010001100010011	
M_3	0000111100001111	

Popcount

3) Bits innerhalb 8er Blöcke addieren, dazu 4er-Maske M_3 nutzen.

	15	0
x	0010001100010011	
M_3	0000111100001111	
$x \,\&\, M_3$	0000001100000011	

Popcount

3) Bits innerhalb 8er Blöcke addieren, dazu 4er-Maske M_3 nutzen.

	15	0
x	0010001100010011	
M_3	0000111100001111	
$x \,\&\, M_3$	0000001100000011	
$+(x \gg 4) \,\&\, M_3$	0000001000000001	

Popcount

3) Bits innerhalb 8er Blöcke addieren, dazu 4er-Maske M_3 nutzen.

	15	0
x	0010001100010011	
M_3	0000111100001111	
$x \,\&\, M_3$	0000001100000011	
$+(x \gg 4) \,\&\, M_3$	0000001000000001	
$x =$	0000010100000100	

Popcount

4) Die letzten beiden Blöcke addieren, dazu 8er-Maske M_4 nutzen.

$$\begin{array}{cc} & \overset{15}{} \qquad\qquad\qquad\qquad \overset{0}{} \\ x & 0000010100000100 \end{array}$$

Popcount

4) Die letzten beiden Blöcke addieren, dazu 8er-Maske M_4 nutzen.

	15	0
x	0000010100000100	
M_4	0000000011111111	

Popcount

4) Die letzten beiden Blöcke addieren, dazu 8er-Maske M_4 nutzen.

	15 … 0
x	0000010100000100
M_4	0000000011111111
$x \,\&\, M_4$	0000000000000100

Popcount

4) Die letzten beiden Blöcke addieren, dazu 8er-Maske M_4 nutzen.

	15 0
x	0000010100000100
M_4	0000000011111111
$x \,\&\, M_4$	0000000000000100
$+(x \gg 8)$	0000000000000101

Popcount

4) Die letzten beiden Blöcke addieren, dazu 8er-Maske M_4 nutzen.

	15 0
x	0000010100000100
M_4	0000000011111111
$x \,\&\, M_4$	0000000000000100
$+(x \gg 8)$	0000000000000101
$x =$	0000000000001001

Popcount

Popcount hat Laufzeit $\mathcal{O}(\log W)$. Man kann dies in C-Code mit einigen Tricks noch effizienter codieren[1]; hier wird $W = 64$ angenommen.

```
1  const uint64 M1   = 0x5555555555555555;
2  const uint64 M2   = 0x3333333333333333;
3  const uint64 M3   = 0x0f0f0f0f0f0f0f0f;
4  const uint64 H256 = 0x0101010101010101;
5  int popcount(uint64 x) {
6      x -= (x >> 1) & M1;
7      x = (x & M2) + ((x >> 2) & M2);
8      x = (x + (x >> 4)) & M3;
9      return (x * H256) >> 56;
10 }
```

[1] Quelle: http://en.wikipedia.org/wiki/Hamming_weight

Rank-Datenstruktur

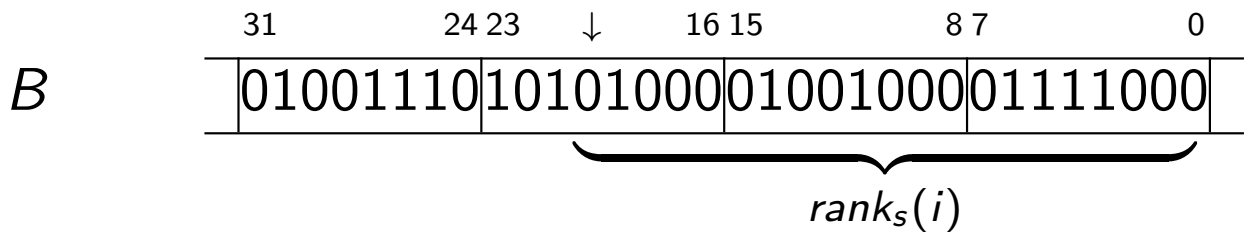
- Popcount in konstanter Zeit berechenbar, da W konstant ist.
- Popcount-Anfrage auf $s[:i]$ wäre also in $\mathcal{O}(n/W)$ berechenbar.

Rank-Datenstruktur

- Popcount in konstanter Zeit berechenbar, da W konstant ist.
- Popcount-Anfrage auf $s[:i]$ wäre also in $\mathcal{O}(n/W)$ berechenbar.
- Durch Verwendung von Hilfsstrukturen wird nur noch konstante Zeit benötigt.
- Solch eine Datenstruktur wird *rank*-Datenstruktur genannt.
- Anzahl der 1en in $s[:i]$ wird mit $rank_s(i)$ bezeichnet.

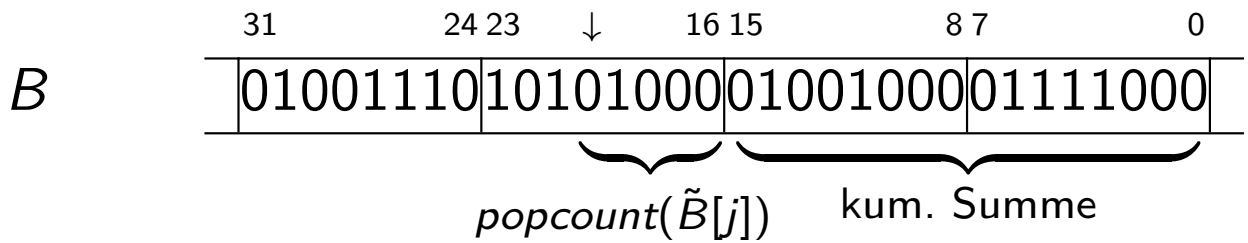
Rank-Datenstruktur

Beispiel: gegeben Bitvektor s bzw. Wortvektor B, $W = 8, n = 32$, $i = 20$; gesucht $rank_s(i)$.



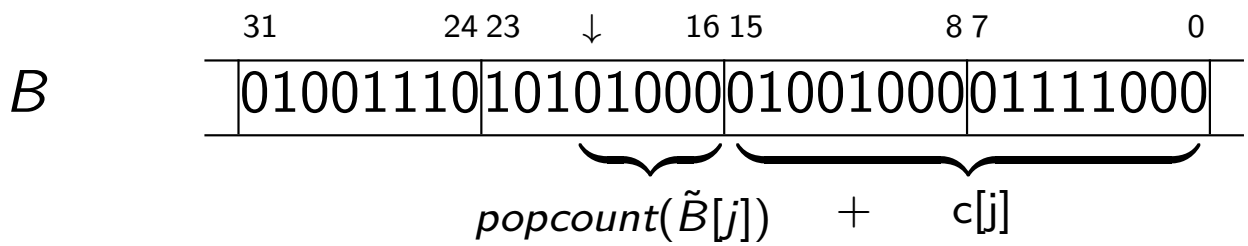
Rank-Datenstruktur

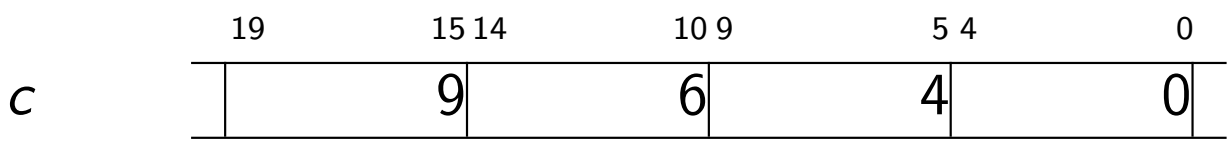
Beispiel: gegeben Bitvektor s bzw. Wortvektor B, $W = 8, n = 32$, $i = 20$; gesucht $rank_s(i)$.



Rank-Datenstruktur

Beispiel: gegeben Bitvektor s bzw. Wortvektor B, $W = 8, n = 32$, $i = 20$; gesucht $rank_s(i)$.





Rank-Datenstruktur

Um die ersten $k = (i \ \% \ W) + 1$ Bits in $B[j] = 00101000$ zu zählen, gibt es mehrere Möglichkeiten:

- $\tilde{B}[j] = B[j] \ \& \ ((1 \ll k) - 1)$

Rank-Datenstruktur

Um die ersten $k = (i \ \% \ W) + 1$ Bits in $B[j] = 00101000$ zu zählen, gibt es mehrere Möglichkeiten:

- $\tilde{B}[j] = B[j] \ \& \ ((1 \ll k) - 1)$, *Achtung:* $1 \ll W$ möglich

Rank-Datenstruktur

Um die ersten $k = (i \ \% \ W) + 1$ Bits in $B[j] = 00101000$ zu zählen, gibt es mehrere Möglichkeiten:

- $\tilde{B}[j] = B[j] \ \& \ ((1 \ll k) - 1)$, *Achtung:* $1 \ll W$ möglich
- $\tilde{B}[j] = B[j] \ \& \ (\sim 0 \gg (W - k))$

Rank-Datenstruktur

Um die ersten $k = (i \ \% \ W) + 1$ Bits in $B[j] = 00101000$ zu zählen, gibt es mehrere Möglichkeiten:

- $\tilde{B}[j] = B[j] \ \& \ ((1 \ll k) - 1)$, *Achtung:* $1 \ll W$ möglich
- $\tilde{B}[j] = B[j] \ \& \ (\sim 0 \gg (W - k))$
- $\tilde{B}[j] = B[j] \ll (W - k)$, besser

Rank-Datenstruktur

Um die ersten $k = (i \ \% \ W) + 1$ Bits in $B[j] = 00101000$ zu zählen, gibt es mehrere Möglichkeiten:

- $\tilde{B}[j] = B[j] \ \& \ ((1 \ll k) - 1)$, *Achtung:* $1 \ll W$ möglich
- $\tilde{B}[j] = B[j] \ \& \ (\sim 0 \gg (W - k))$
- $\tilde{B}[j] = B[j] \ll (W - k)$, besser

Kompletter Aufruf:
$rank_s(i) = c[j] + popcount\big(B[j] \ll (W - (i \ \& \ m) - 1)\big)$,
wobei $j = i \gg W, m = (1 \ll w) - 1$.

Rank-Datenstruktur

Der Aufbau sei folgendermaßen definiert:

$$c[0] := 0$$
$$c[j] := c[j-1] + popcount(B[j]) \text{ für alle } 1 \leq j < \lceil n/W \rceil.$$

Das Hilfsarray der kumulierten Summen c hat folgende Eigenschaften:

- Es besitzt (wie Array B) $\lceil n/W \rceil$ Felder.
- Ein Feld braucht maximal $\lceil \log n \rceil$ Bits.
- Es benötigt also insgesamt $\lceil n/W \rceil \cdot \lceil \log n \rceil$ Bits.

Rank-Datenstruktur

Der Aufbau sei folgendermaßen definiert:

$$c[0] := 0$$
$$c[j] := c[j-1] + popcount(B[j]) \text{ für alle } 1 \leq j < \lceil n/W \rceil.$$

Das Hilfsarray der kumulierten Summen c hat folgende Eigenschaften:

- Es besitzt (wie Array B) $\lceil n/W \rceil$ Felder.
- Ein Feld braucht maximal $\lceil \log n \rceil$ Bits.
- Es benötigt also insgesamt $\lceil n/W \rceil \cdot \lceil \log n \rceil$ Bits.
- Problem: Platz wird verschwendet, wenn $\lceil \log n \rceil < W$ ist.

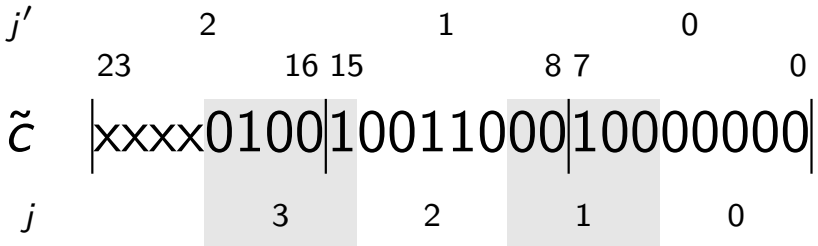
Rank-Datenstruktur

Hilfsarray c muss in einem Array $\tilde{c}$ mit Wortlänge W gespeicht werden. Aber wie sollen die Einträge aus $\tilde{c}$ ausgelesen werden?

Rank-Datenstruktur

Hilfsarray c muss in einem Array $\tilde{c}$ mit Wortlänge W gespeicht werden. Aber wie sollen die Einträge aus $\tilde{c}$ ausgelesen werden?

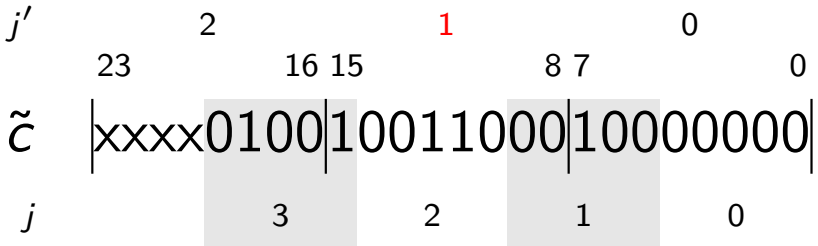
Beispiel: gegeben sei Array $\tilde{c}$ mit $W = 8, w = \lceil \log W \rceil = 3$, $W' = \lceil \log n \rceil = 5$; gesucht ist $c[2]$.



Rank-Datenstruktur

Hilfsarray c muss in einem Array $\tilde{c}$ mit Wortlänge W gespeicht werden. Aber wie sollen die Einträge aus $\tilde{c}$ ausgelesen werden?

Beispiel: gegeben sei Array $\tilde{c}$ mit $W = 8, w = \lceil \log W \rceil = 3$, $W' = \lceil \log n \rceil = 5$; gesucht ist $c[2]$.

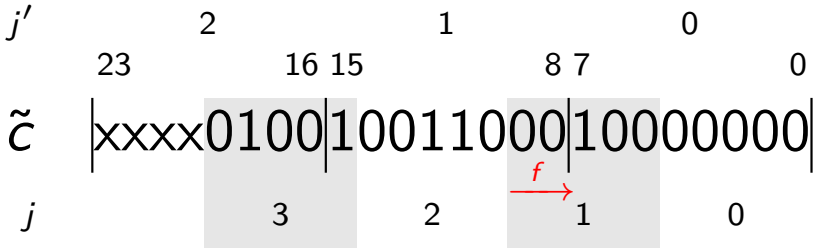


1) Index j' aus j ermitteln:
$j' = (j \cdot W') \gg w = 1$

Rank-Datenstruktur

Hilfsarray c muss in einem Array $\tilde{c}$ mit Wortlänge W gespeicht werden. Aber wie sollen die Einträge aus $\tilde{c}$ ausgelesen werden?

Beispiel: gegeben sei Array $\tilde{c}$ mit $W = 8, w = \lceil \log W \rceil = 3$, $W' = \lceil \log n \rceil = 5$; gesucht ist $c[2]$.

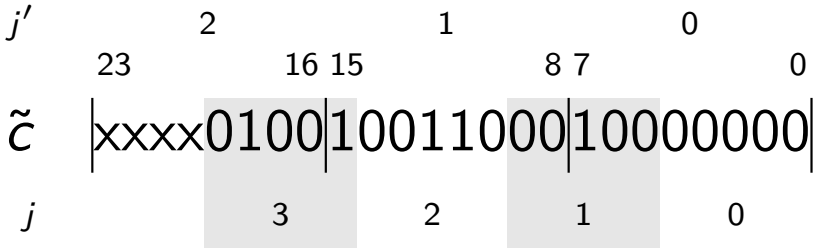


2) Rechtsverschiebung f von $\tilde{c}[j']$ berechnen:
$f = (j \cdot W') \,\&\, ((1 \ll w) - 1) = 2$

Rank-Datenstruktur

Hilfsarray c muss in einem Array $\tilde{c}$ mit Wortlänge W gespeicht werden. Aber wie sollen die Einträge aus $\tilde{c}$ ausgelesen werden?

Beispiel: gegeben sei Array $\tilde{c}$ mit $W = 8, w = \lceil \log W \rceil = 3$, $W' = \lceil \log n \rceil = 5$; gesucht ist $c[2]$.



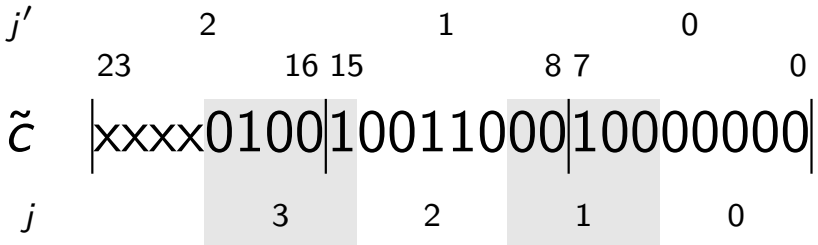
$\tilde{c}[j'] \gg f = 00100110$
$(1 \ll W') - 1 = 00011111$

3) $\tilde{c}[j']$ verschieben und mit 5er Bitmaske maskieren:
$c[j] = (\tilde{c}[j'] \gg f)\ \&\ ((1 \ll W') - 1) = 6$

Rank-Datenstruktur

Hilfsarray c muss in einem Array $\tilde{c}$ mit Wortlänge W gespeicht werden. Aber wie sollen die Einträge aus $\tilde{c}$ ausgelesen werden?

Beispiel: gegeben sei Array $\tilde{c}$ mit $W = 8, w = \lceil \log W \rceil = 3$, $W' = \lceil \log n \rceil = 5$; gesucht ist $c[2]$.

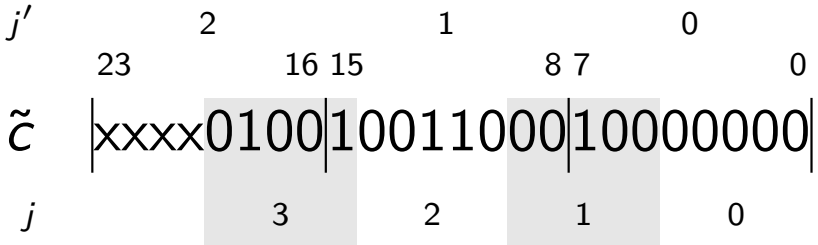


Was ist, wenn $c[j]$ über zwei Wörter geschrieben ist?

Rank-Datenstruktur

Hilfsarray c muss in einem Array $\tilde{c}$ mit Wortlänge W gespeicht werden. Aber wie sollen die Einträge aus $\tilde{c}$ ausgelesen werden?

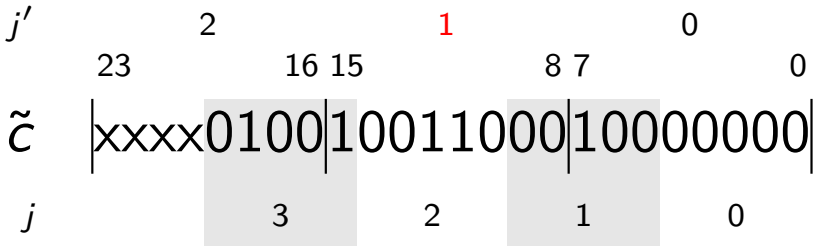
Beispiel: gegeben sei Array $\tilde{c}$ mit $W = 8, w = \lceil \log W \rceil = 3$, $W' = \lceil \log n \rceil = 5$; gesucht ist $c[3]$.



Rank-Datenstruktur

Hilfsarray c muss in einem Array $\tilde{c}$ mit Wortlänge W gespeicht werden. Aber wie sollen die Einträge aus $\tilde{c}$ ausgelesen werden?

Beispiel: gegeben sei Array $\tilde{c}$ mit $W = 8, w = \lceil \log W \rceil = 3$, $W' = \lceil \log n \rceil = 5$; gesucht ist $c[3]$.

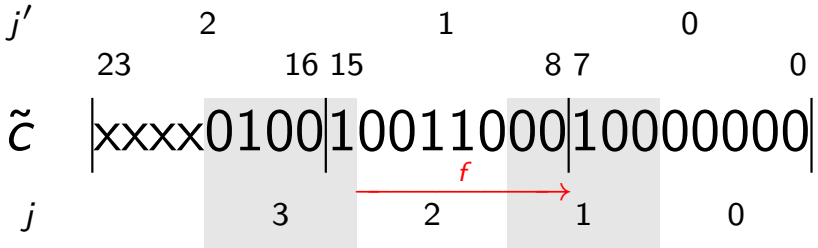


1) Index j' aus j ermitteln:
$j' = (j \cdot W') \gg w = 1$

Rank-Datenstruktur

Hilfsarray c muss in einem Array $\tilde{c}$ mit Wortlänge W gespeicht werden. Aber wie sollen die Einträge aus $\tilde{c}$ ausgelesen werden?

Beispiel: gegeben sei Array $\tilde{c}$ mit $W = 8, w = \lceil \log W \rceil = 3$, $W' = \lceil \log n \rceil = 5$; gesucht ist $c[3]$.

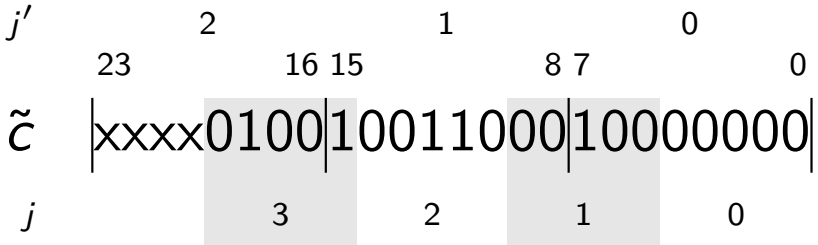


2) Rechtsverschiebung f von $\tilde{c}[j']$ berechnen:
$f = (j \cdot W') \,\&\, ((1 \ll w) - 1) = 7$

Rank-Datenstruktur

Hilfsarray c muss in einem Array $\tilde{c}$ mit Wortlänge W gespeicht werden. Aber wie sollen die Einträge aus $\tilde{c}$ ausgelesen werden?

Beispiel: gegeben sei Array $\tilde{c}$ mit $W = 8, w = \lceil \log W \rceil = 3$, $W' = \lceil \log n \rceil = 5$; gesucht ist $c[3]$.



$\tilde{c}[j'] \gg f = 00000001$

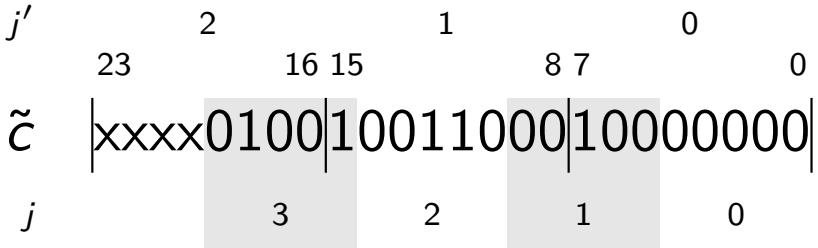
$\tilde{c}[j'+1] \ll (W-f) = xxx01000$

3) $\tilde{c}[j']$ und $\tilde{c}[j'+1]$ verschieben und verodern:
$e = (\tilde{c}[j'] \gg f) \mid (\tilde{c}[j'+1] \ll (W-f))$

Rank-Datenstruktur

Hilfsarray c muss in einem Array $\tilde{c}$ mit Wortlänge W gespeicht werden. Aber wie sollen die Einträge aus $\tilde{c}$ ausgelesen werden?

Beispiel: gegeben sei Array $\tilde{c}$ mit $W = 8, w = \lceil \log W \rceil = 3$, $W' = \lceil \log n \rceil = 5$; gesucht ist $c[3]$.



$e = \text{xxx01001}$

$(1 \ll W') - 1 = 00011111$

4) Ergebnis wieder mit 5er Bitmaske maskieren:
$c[j] = e \;\&\; ((1 \ll W') - 1) = 9$

Rank-Datenstruktur

Durch geschicktes Speichern von Hilfsarray c lässt sich eine Verschwendung von Speicherplatz vermeiden. Aber lässt sich auch Speicherplatz einsparen?

Rank-Datenstruktur

Durch geschicktes Speichern von Hilfsarray c lässt sich eine Verschwendung von Speicherplatz vermeiden. Aber lässt sich auch Speicherplatz einsparen?

Beobachtung: pro Feld in B erhöht sich die Summe in c höchstens um den Wert W (w Bits), jedoch muss diese Erhöhung im Feld mit Größe $\lceil \log n \rceil > w$ gespeichert werden.

Rank-Datenstruktur

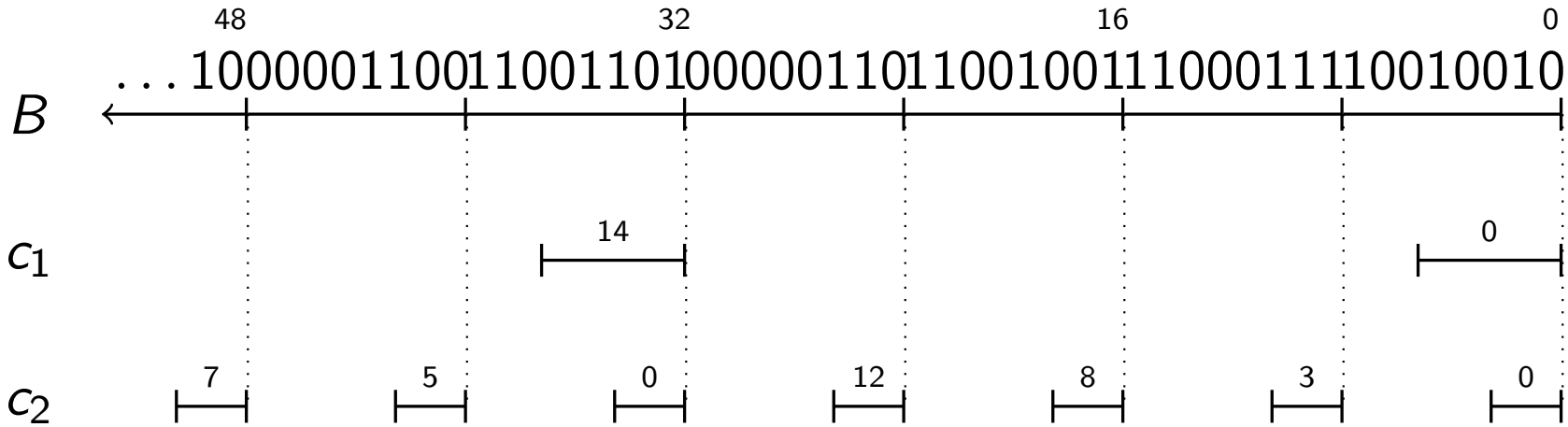
Durch geschicktes Speichern von Hilfsarray c lässt sich eine Verschwendung von Speicherplatz vermeiden. Aber lässt sich auch Speicherplatz einsparen?

Beobachtung: pro Feld in B erhöht sich die Summe in c höchstens um den Wert W (w Bits), jedoch muss diese Erhöhung im Feld mit Größe $\lceil \log n \rceil > w$ gespeichert werden.

Idee: Hilfsarray c samplen, zweite Ebene c_2 einführen, die mit weniger Bits auskommt.

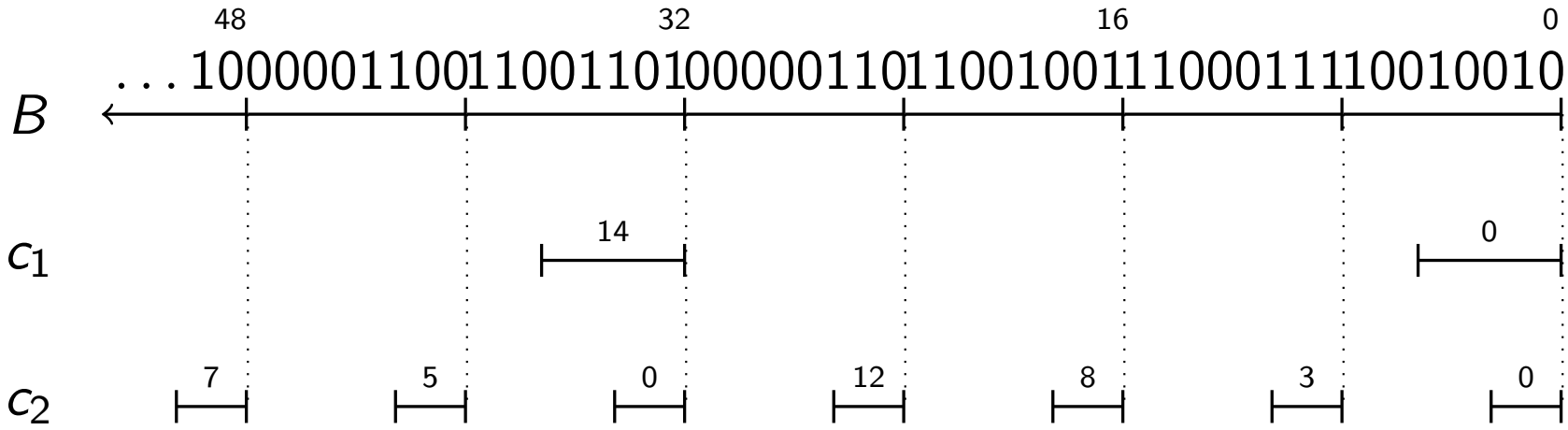
Rank-Datenstruktur

Beispiel: gegeben Bitvektor s bzw. Wortvektor B, $W = 8$, $n = |s|$, Samplefaktor $2^k = 4$.



Rank-Datenstruktur

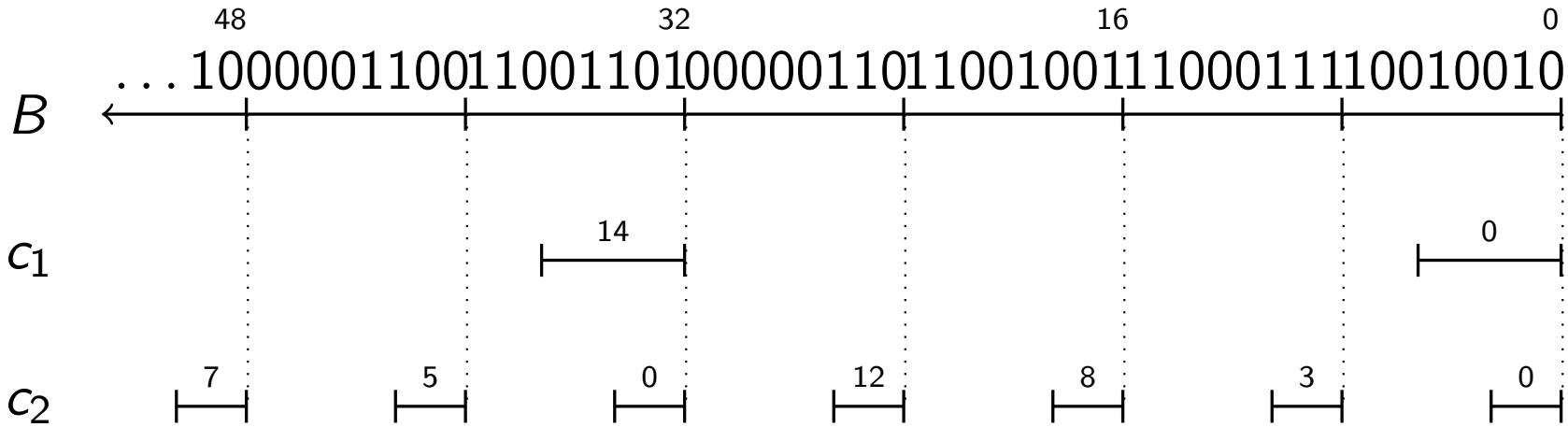
Beispiel: gegeben Bitvektor s bzw. Wortvektor B, $W = 8$, $n = |s|$, Samplefaktor $2^k = 4$.



- c_1 speichert weiterhin die kumulative Summe für den kompletten Präfix von B.
- c_1 wird gesamplet, hat also nur noch $\lceil n/(W \cdot 2^k) \rceil$ Einträge.

Rank-Datenstruktur

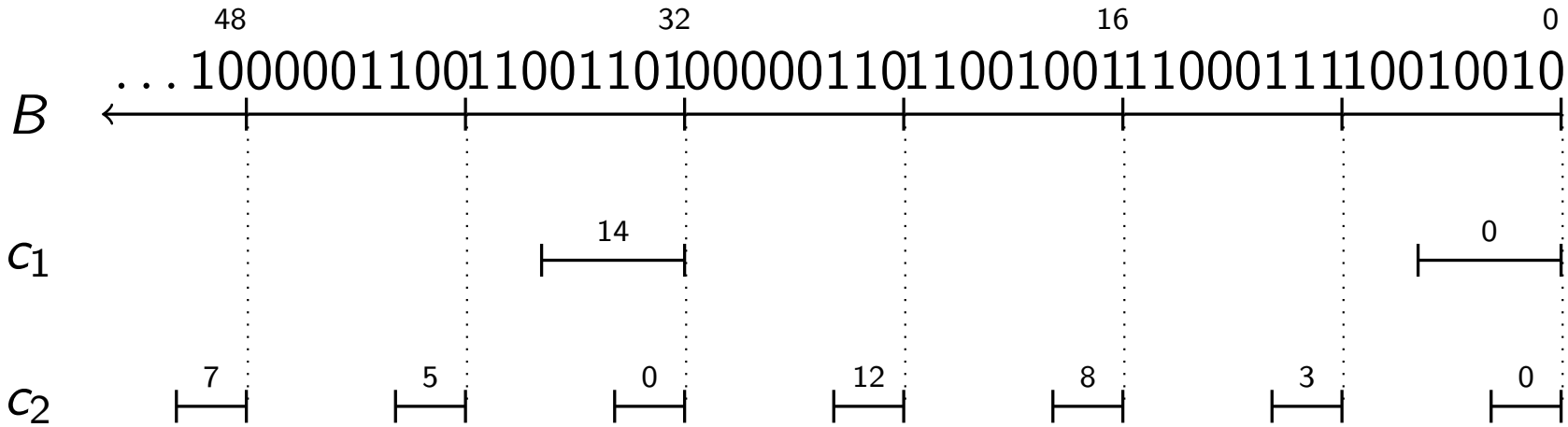
Beispiel: gegeben Bitvektor s bzw. Wortvektor B, $W = 8$, $n = |s|$, Samplefaktor $2^k = 4$.



- c_2 speichert nur kumulative Summen für 2^k-lange Teilstrings von B, beginnt immer wieder bei 0.
- c_2 braucht nur $k + w$ Bits pro Feld zu speichern.

Rank-Datenstruktur

Beispiel: gegeben Bitvektor s bzw. Wortvektor B, $W = 8$, $n = |s|$, Samplefaktor $2^k = 4$.



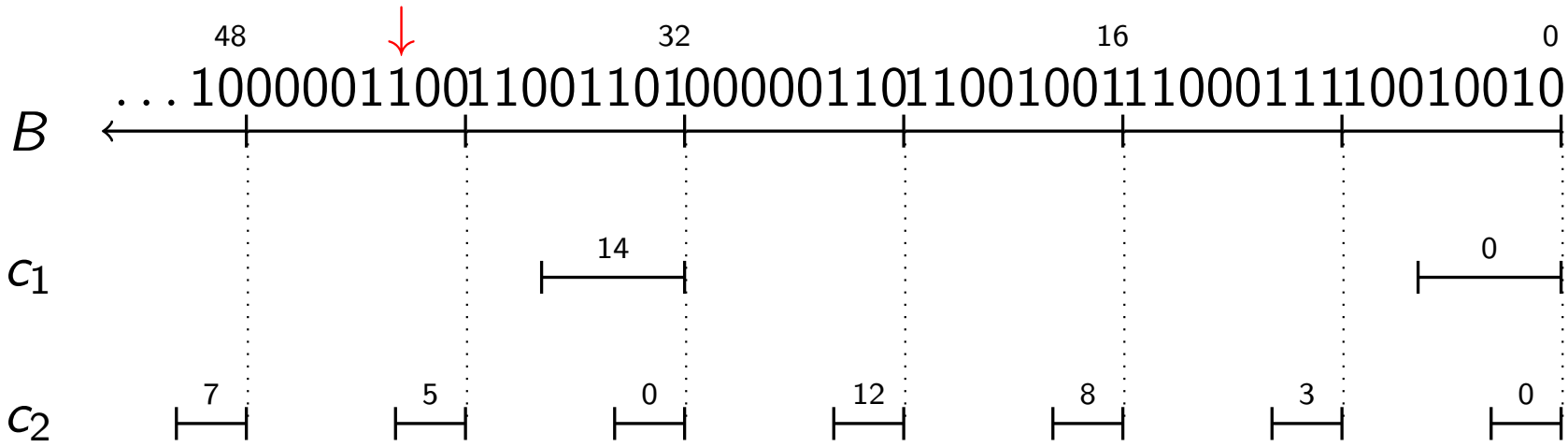
Rank-Aufruf muss nur leicht modifiziert werden:

$$rank_s(i) = c_1[j \gg k] + c_2[j] + popcount(\dots)$$

mit $j = i \gg w$.

Rank-Datenstruktur

Beispiel: gegeben Bitvektor s bzw. Wortvektor B, $W = 8$, $n = |s|$, Samplefaktor $2^k = 4$.



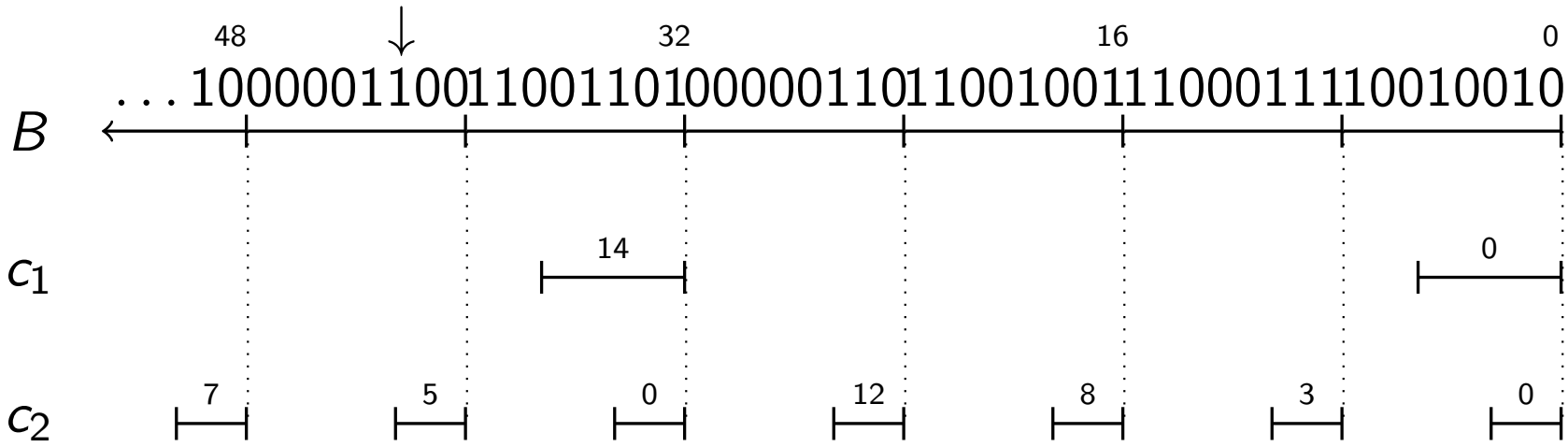
Rank-Aufruf muss nur leicht modifiziert werden, Beispiel $i = 42$:

$$rank_s(42) = c_1[j \gg k] + c_2[j] + popcount(\dots)$$

mit $j = i \gg w$.

Rank-Datenstruktur

Beispiel: gegeben Bitvektor s bzw. Wortvektor B, $W = 8$, $n = |s|$, Samplefaktor $2^k = 4$.



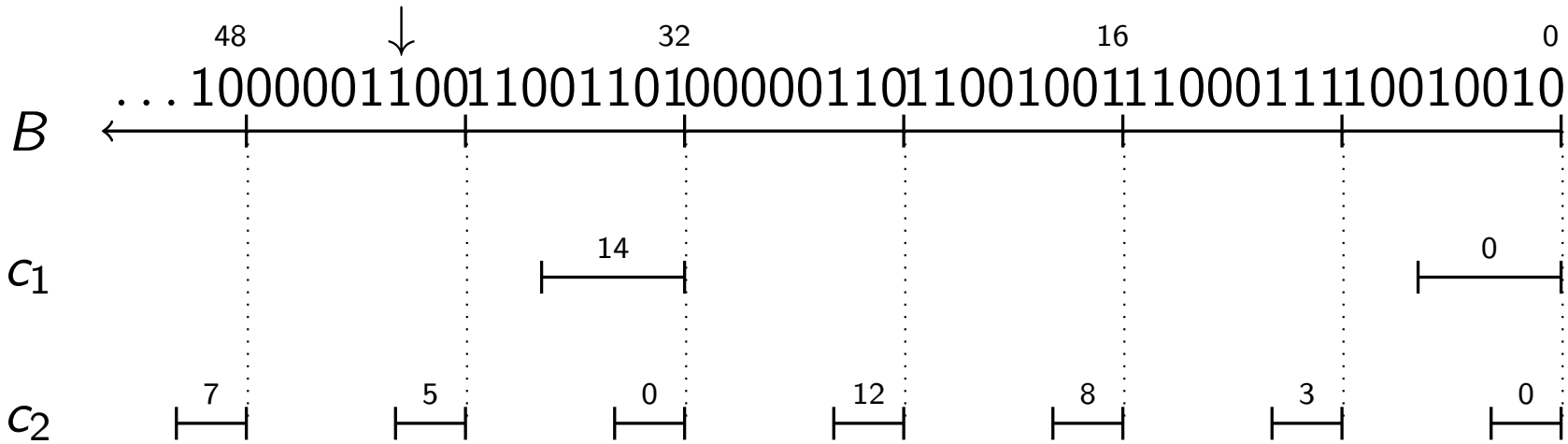
Rank-Aufruf muss nur leicht modifiziert werden, Beispiel $i = 42$:

$$rank_s(42) = c_1[j \gg k] + c_2[j] + popcount(\dots)$$

mit $j = 42 \gg w$.

Rank-Datenstruktur

Beispiel: gegeben Bitvektor s bzw. Wortvektor B, $W = 8$, $n = |s|$, Samplefaktor $2^k = 4$.



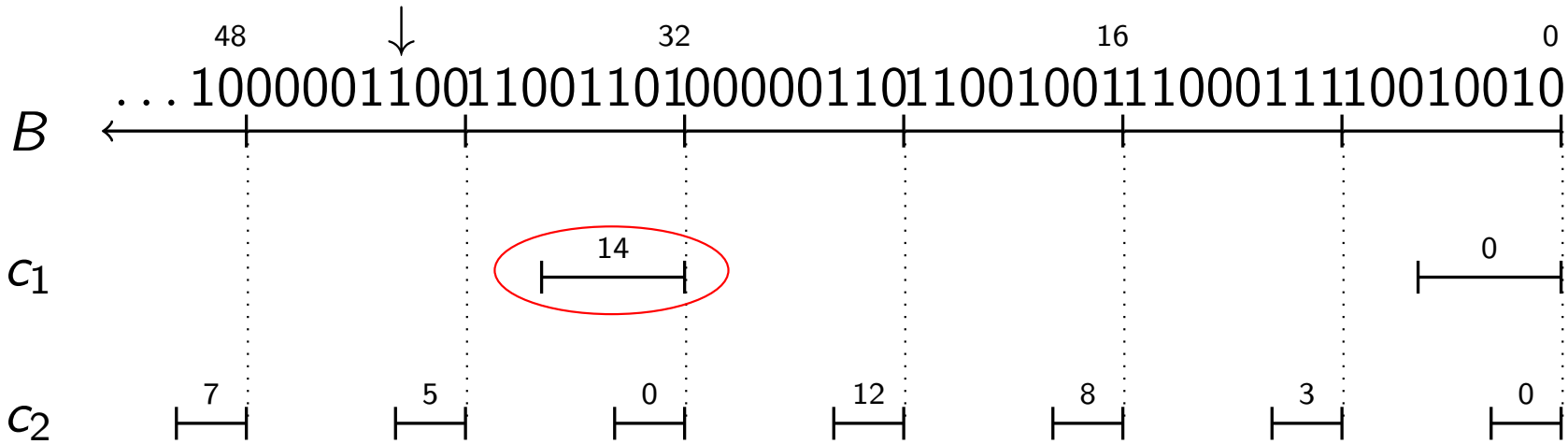
Rank-Aufruf muss nur leicht modifiziert werden, Beispiel $i = 42$:

$$rank_s(42) = c_1[5 \gg k] + c_2[5] + popcount(\dots)$$

mit $j = 42 \gg w = 5$.

Rank-Datenstruktur

Beispiel: gegeben Bitvektor s bzw. Wortvektor B, $W = 8$, $n = |s|$, Samplefaktor $2^k = 4$.



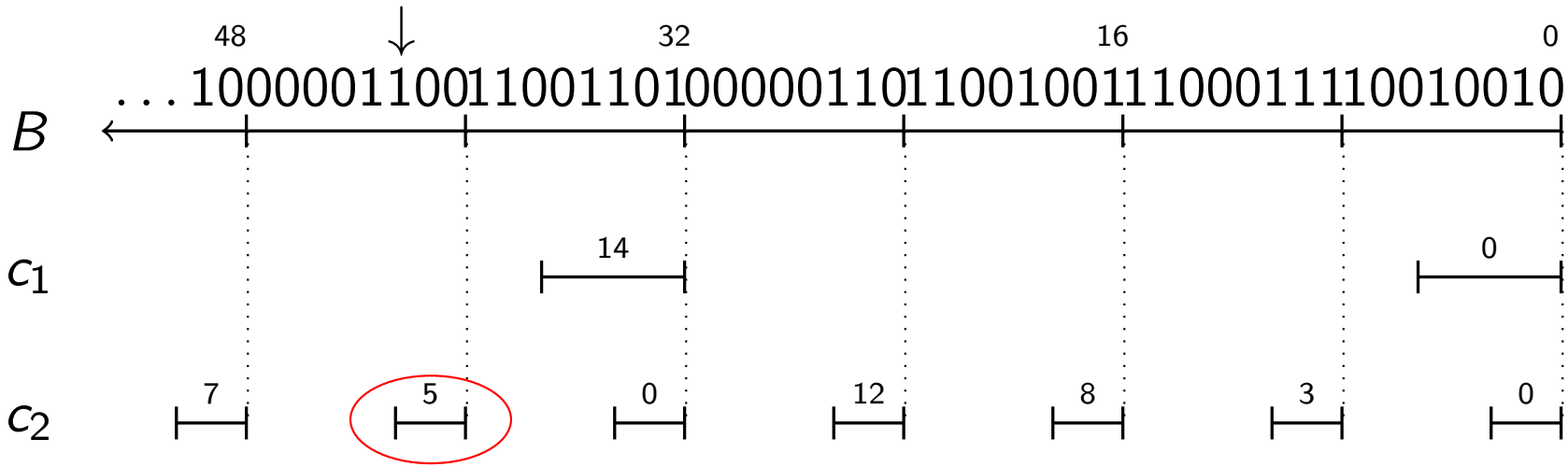
Rank-Aufruf muss nur leicht modifiziert werden, Beispiel $i = 42$:

$$rank_s(42) = c_1[5 \gg k] + c_2[5] + popcount(\dots)$$

mit $j = 42 \gg w = 5$.

Rank-Datenstruktur

Beispiel: gegeben Bitvektor s bzw. Wortvektor B, $W = 8$, $n = |s|$, Samplefaktor $2^k = 4$.



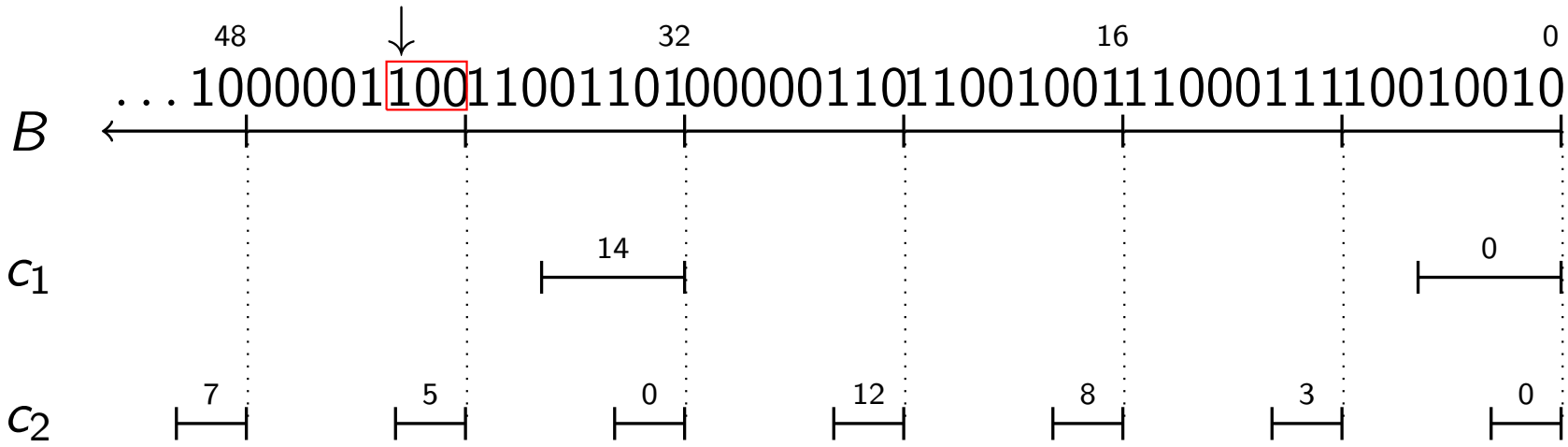
Rank-Aufruf muss nur leicht modifiziert werden, Beispiel $i = 42$:

$$rank_s(42) = 14 + c_2[5] + popcount(\dots)$$

mit $j = 42 \gg w = 5$.

Rank-Datenstruktur

Beispiel: gegeben Bitvektor s bzw. Wortvektor B, $W = 8$, $n = |s|$, Samplefaktor $2^k = 4$.



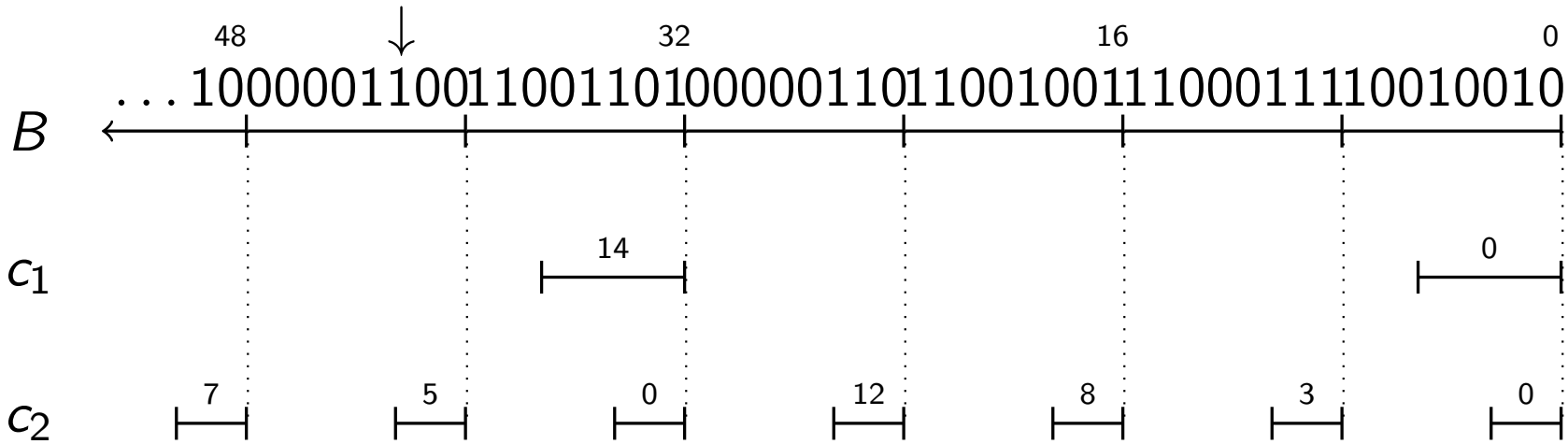
Rank-Aufruf muss nur leicht modifiziert werden, Beispiel $i = 42$:

$$rank_s(42) = 14 + 5 + popcount(\ldots)$$

mit $j = 42 \gg w = 5$.

Rank-Datenstruktur

Beispiel: gegeben Bitvektor s bzw. Wortvektor B, $W = 8$, $n = |s|$, Samplefaktor $2^k = 4$.



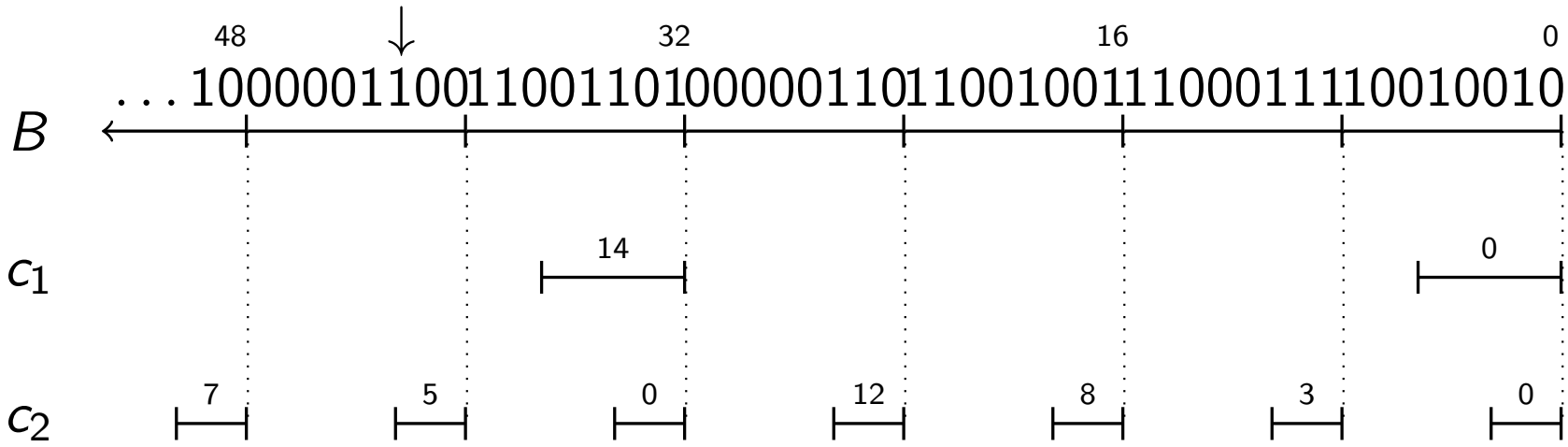
Rank-Aufruf muss nur leicht modifiziert werden, Beispiel $i = 42$:

$$rank_s(42) = 14 + 5 + 1$$

mit $j = 42 \gg w = 5$.

Rank-Datenstruktur

Beispiel: gegeben Bitvektor s bzw. Wortvektor B, $W = 8$, $n = |s|$, Samplefaktor $2^k = 4$.



Rank-Aufruf muss nur leicht modifiziert werden, Beispiel $i = 42$:

$$rank_s(42) = 20$$

mit $j = 42 \gg w = 5$.

Rank-Datenstruktur

Vergleich des Speicherverbrauchs beider Hilfsstruktur-Varianten:

- eine Ebene: $\lceil n/W \rceil \cdot \lceil \log n \rceil$
- zwei Ebenen: $\underbrace{\lceil n/(W \cdot 2^k) \rceil \cdot \lceil \log n \rceil}_{c_1} + \underbrace{\lceil n/W \rceil \cdot (k+w)}_{c_2}$

Die Variante mit zwei Ebenen lohnt sich, wenn $n \gtrsim 2^k \cdot W$ ist.

Zusammenfassung

- Bitsequenzen sind die einfachste Form von Sequenzen.
- Mit der Funktion *popcount* kann die Anzahl der 1en eines Wortes in $\mathcal{O}(\log W)$ berechnet werden.
- Die Rank-Datenstruktur erweitert das Zählen der 1en auf ein Wortarray.
- Rank wird u.a. eingesetzt, um große dünnbesiedelte Arrays auf kleine komplettbefüllte Arrays zu mappen.
- Arbeiten auf Bitebene wird bei einigen Sequenzalgorithmen eingesetzt.