Stefan Kurtz **Universität Hamburg**
Christina de Bruyn Kops **Zentrum für Bioinformatik**

Grundlagen der Sequenzanalyse
Wintersemester 2016
Übungen zur Vorlesung: Ausgabe am 18.10.2016

Punkteverteilung: Aufgabe 1.1: 4 Punkte, Aufgabe 1.2: 4 Punkte, Aufgabe 1.3: 2 Punkte

Abgabe bis zum 24.10. Hinweis zu Aufgaben 1.2 und 1.3: Anstatt Pseudocode kann auch lauffähiger Programmcode abgegeben werden.

Aufgabe 1.1 Berechnen Sie folgende Distanzen:

1. Hamming Distanz: Berechnen Sie, falls möglich, die Hamming-Distanz für alle Paare der folgenden Sequenzen: $s_1 = acgatactag$, $s_2 = aggtcattga$, $s_3 = aggcattga$, $s_4 = cgatactaga$.
2. Euklidische Distanz und Block-Distanz: Berechnen Sie jeweils die Euklidische Distanz und die Block-Distanz für alle Paare der folgenden drei Vektoren der Länge 7:

   $v_1 = (1, 4, 3, 9, 1, 2, 5)$,
   $v_2 = (6, 4, 11, -9, -4, 8, 7)$,
   $v_3 = (5, 1, 4, 3, 9, 1, 2)$.

Aufgabe 1.2 Bei der Analyse von Sequenzen ist es häufig nötig, alle Substrings w der Länge k (k-Worte) einer Sequenz s aufzuzählen. Man kann sich dies als Bewegen eines „Fensters“ der Länge k über s vorstellen, so dass der jeweils in Schritt i aktuell betrachtete Inhalt sich in einem Puffer w befindet.

Beispiel für $k = 5$ und $s = gagtccgatggcgttggat$:

$$\overbrace{\underbrace{gagtc}_{w_1}}^{i=1} cgatggcgttggat \rightarrow g \overbrace{\underbrace{agtcc}_{w_2}}^{i=2} gatggcgttggat \rightarrow ga \overbrace{\underbrace{gtccg}_{w_3}}^{i=3} atggcgttggat \rightarrow \ldots$$

Dabei ist w_i das k-Wort an Position i.

Bei langen Sequenzen ist es manchmal nicht möglich, die gesamte Sequenz in den Arbeitsspeicher zu laden. Dann ist im Allgemeinen nur ein sequenzieller Zugriff (etwa über einen Massenspeicher) möglich, bei dem sich das jeweils nächste Zeichen der Sequenz über eine gegebene Funktion $getnext(s)$ lesen läßt. Ein Zugriff auf vorher gelesene Zeichen ist also nur möglich, wenn man diese selbst speichert. Beschreiben Sie einen Algorithmus, der die k-Worte w_i von s ausgibt, ohne jedoch s dabei komplett in den Speicher zu laden. Der verwendete Speicherplatz soll auf k Zeichen beschränkt sein. Falls Pseudocode abgegeben wird, muß die Funktion $getnext(s)$ nicht beschrieben werden und kann als gegeben hingenommen werden.

Aufgabe 1.3 Man unterscheidet bei doppelsträngiger DNA zwischen der Basensequenz auf dem *plus*- und dem *minus*-Strang. Durch Basenpaarung bedingt sind diese *revers komplementär* zueinander. Sei $\mathcal{A}_{dna} = \{a, c, g, t\}$ das DNA-Alphabet. Das *Komplement* $\overline{x}$ eines Zeichens $x \in \mathcal{A}_{dna}$ ist dann definiert als $\overline{a} = t$, $\overline{t} = a$, $\overline{c} = g$ und $\overline{g} = c$.

Für eine DNA Sequenz $s \in \mathcal{A}_{dna}^*$ der Länge n ist das *reverse Komplement* $\overleftarrow{s}$ von s definiert als $\overleftarrow{s} := \overline{s[n]}\ \overline{s[n-1]}\ \ldots\ \overline{s[2]}\ \overline{s[1]}$. Zum Beispiel ist $\overleftarrow{gagctgaa} = ttcagctc$.

Beschreiben Sie einen Algorithmus, der zu einer gegebenen Sequenz das reverse Komplement bestimmt, ohne mehr Speicherplatz als die ursprüngliche Sequenz zu verbrauchen, d.h. es soll keine Kopie der Sequenz angelegt werden. (Konstanter Speicherplatz, etwa zum temporären Zwischenspeichern einzelner Zeichen, ist zulässig.)

Die Lösungen zu diesen Aufgaben werden am 25.10.2016 besprochen.

Lösung zu Aufgabe 1.1:

1. Hamming Distanz:

   $f(s_1, s_2) = f(acgatactag, aggtcattga) = |\{2, 4, 5, 7, 9, 10\}| = 6$

   $f(s_1, s_3) = \perp$ (undefiniert, da $|s_1| \neq |s_3|$)

   $f(s_1, s_4) = |\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}| = 10$

   $f(s_2, s_3) = \perp$ (undefiniert, da $|s_2| \neq |s_3|$)

   $f(s_2, s_4) = |\{1, 3, 5, 6, 8\}| = 5$

   $f(s_3, s_4) = \perp$ (undefiniert, da $|s_3| \neq |s_4|$)

2. Euklidische Distanz & Block Distanz:

   Euklidische Distanz:

$$\begin{aligned} f(v_1, v_2) &= \sqrt{\sum_{i=1}^{n} (v_1[i] - v_2[i])^2} \\ &= \sqrt{(1-6)^2 + (4-4)^2 + (3-11)^2 + (9-(-9))^2 + (1-(-4))^2 + (2-8)^2 + (5-7)^2} \\ &= \sqrt{25 + 0 + 64 + 324 + 25 + 36 + 4} \\ &= \sqrt{478} \\ &= 21{,}86 \end{aligned}$$

   $f(v_1, v_3) = \sqrt{136} = 11{,}66$

   $f(v_2, v_3) = \sqrt{446} = 21{,}11$

   Block Distanz:

$$\begin{aligned} f(v_1, v_2) &= \sum_{i=1}^{n} |v_1[i] - v_2[i]| \\ &= |1-6| + |4-4| + |3-11| + |9-(-9)| + |1-(-4)| + |2-8| + |5-7| \\ &= 5 + 0 + 8 + 18 + 5 + 6 + 2 \\ &= 44 \end{aligned}$$

   $f(v_1, v_3) = 26$

   $f(v_2, v_3) = 48$

Lösung zu Aufgabe 1.2:

Folgender Algorithmus speichert jeweils ein k-Wort von s in dem Ringpuffer w und ruft für jedes die Funktion $printwindow()$ auf, die den Inhalt ausgibt. Das Symbol $\perp$ bezeichnet damit den Rückgabewert von $getnext()$, wenn kein Zeichen mehr zu lesen ist.

```
1: procedure ENUMERATEKMERS((s, k))
2:     f ← 1
3:     n ← 1
4:     c = getnext(s)
```

```
5:      full = false
6:      while c ≠ ⊥ do
7:          w[f] ← c
8:          f ← f + 1
9:          if f > k then
10:             f ← 1
11:             full = true
12:         end if
13:         if full = true then
14:             printwindow(w, f, k)
15:         end if
16:         c = getnext(s)
17:     end while
18: end procedure
```

Diese Funktion gibt den Inhalt des Ringpuffers w mit der Länge n beginnend ab der Position f aus. Alternativ könnte man hier auch andere Funktionalität aufrufen.

```
1: procedure PRINTWINDOW((w, f, k))
2:     for i ← 0 to k − 1 do
3:         if f + i ≤ k then
4:             print w[f + i]
5:         else
6:             print w[f + i − k]
7:         end if
8:     end for
9: end procedure
```

Hier beispielsweise C-Code, der eine mögliche Implementierung darstellt:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void print_buf(char *buffer, unsigned long firstpos,
               unsigned long windowsize)
{
  unsigned long i;
  for (i = 0; i < windowsize; i++) {
    putchar(buffer[(firstpos + i) % windowsize]);
  }
}

int main(int argc, const char *argv[])
{
  unsigned long firstpos, bufsize, currentpos, windowschecked = 0;
  unsigned long windowsize = 5,
                startpos = 0,
                endpos = strlen(argv[1]);
  char currentchar, *buffer;
  buffer = malloc(sizeof(char) * windowsize);
  firstpos = bufsize = 0;
  for (currentpos=startpos; currentpos < endpos; currentpos++)
  {
    currentchar = argv[1][currentpos];
    if (bufsize < windowsize)
    {
      buffer[bufsize++] = currentchar;
```

```
    } else
    {
      buffer[firstpos++] = currentchar;
      if (firstpos == windowsize)
      {
        firstpos = 0;
      }
    }
    if (bufsize == windowsize)
    {
      print_buf(buffer, firstpos, windowsize);
      printf("\n");
      windowschecked++;
    }
  }
  free(buffer);
}
```

Lösung zu Aufgabe 1.3:

1: **procedure** REVERSE_COMPLEMENT((s))
2: $f \leftarrow 1$
3: $l \leftarrow |s|$
4: **while** $f \leq l$ **do**
5: $t \leftarrow s[f]$
6: $s[f] \leftarrow \overline{s[l]}$
7: $s[l] \leftarrow \overline{t}$
8: $f \leftarrow f + 1$
9: $l \leftarrow l - 1$
10: **end while**
11: **end procedure**

Eine Beispielimplementierung in C findet man in der Datei `src/extended/reverse.c` des *GenomeTools*-Softwarepakets
(`http://genometools.org`).

Beispiel für Pseudocode

Hinweis für LaTeX Nutzer: verwenden Sie die Pakete `algorithm` und `algpseudocode` in der Präambel und schreiben Sie den Pseudocode nach folgendem Schema:

```
\begin{algorithm}
  \caption{Ein Beispiel f\"ur Pseudocode}
  \begin{algorithmic}[1]
    \Require Startbedingungen \Comment{Beschreibung der Eingabe}
    \Ensure Endbedingungen \Comment{Beschreibung der Ausgabe}
    \Statex
    \State \(\mathit{variable}\gets \mathit{value}\) \Comment{Eine Zuweisung}
    \Statex
    \If{Bedingung} \Comment{Bedingte Anweisungen}
      \State (Anweisungen)
    \ElsIf{Alternative Bedingung}
      \State (Anweisungen)
    \Else \Comment{Alternative Anweisungen}
      \State (Anweisungen sonst)
    \EndIf
    \Statex
```

```
    \For{Bedingung, z.B.\@ \(x\in V\)}
      \State (Anweisung)
    \EndFor
    \Statex
    \While{Bedingung, z.B.\@ \(x < y\)}
      \State (Anweisung)
    \EndWhile
  \end{algorithmic}
\end{algorithm}
```

Hier ist das Ergebnis, das man sicher auch anders erzeugen könnte:

Algorithm 1 Ein Beispiel für Pseudocode

Require: Startbedingungen ▷ Beschreibung der Eingabe
Ensure: Endbedingungen ▷ Beschreibung der Ausgabe

1: $variable \leftarrow value$ ▷ Eine Zuweisung

2: **if** Bedingung **then** ▷ Bedingte Anweisungen
3: (Anweisungen)
4: **else if** Alternative Bedingung **then**
5: (Anweisungen)
6: **else** ▷ Alternative Anweisungen
7: (Anweisungen sonst)
8: **end if**

9: **for** Bedingung, z.B. $x \in V$ **do**
10: (Anweisung)
11: **end for**

12: **while** Bedingung, z.B. $x < y$ **do**
13: (Anweisung)
14: **end while**