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

Grundlagen der Sequenzanalyse
Wintersemester 2016/2017
Übungen zur Vorlesung: Ausgabe am 10.01.2017

Punkteverteilung: Aufgabe 11.1: 4 Punkte, Aufgabe 11.2: 4 Punkte, Aufgabe 11.3: 4 Punkte

Abgabe bis zum 16.1.2017

Aufgabe 11.1 Die Funktion *evaluatecrosspoints* für das Linear Space Alignment (siehe Aufgabe 10.1) berechnet das Array C von Kreuzungspunkten. Dieses repräsentiert mindestens ein optimales Alignment der Sequenzen u und v. Die Rekonstruktion eines optimalen Alignments aus C ist jedoch nicht trivial.

Implementieren Sie eine Funktion *reconstructalignment*, die aus dem Array C und den beiden Sequenzen u und v in $O(n)$ Zeit ein optimales Alignment rekonstruiert. Dabei ist $n = |v|$. Sie können wieder das Einheitskostenmodell für die Bewertung der Edit-Operationen vorraussetzen.

Zur Entwicklung der genannten Funktion müssen Sie sich überlegen, für welche Substrings von u und v ein optimales Teilalignment gebildet werden muss, wenn man zwei aufeinander folgende Kreuzungspunkte $C[j]$ und $C[j+1]$ betrachtet. Damit die Alignments mit der Präferenz bei der Berechnung der Kreuzungspunkte (Einfügungen vor Ersetzungen vor Deletionen bezogen auf die Rückwärtskonstruktion eines Alignments) konsistent sind, müssen Sie die Ersetzungsoperationen an den maximal möglichen Positionen anwenden.

Integrieren Sie nun alle Teile der Programmcodes für das linear-space Alignment aus dieser Aufgabe sowie aus Aufgabe 8.2 und 10.1. Alternativ können Sie auch die Kreuzungspunktelisten im Format der Datei `crosspoints.txt` einlesen (siehe Material) und daraus ein Alignment rekonstruieren.

Im Material zu dieser Aufgabe finden Sie in der Datei `sequences.txt` einige Sequenzen und jeweils die entsprechenden Kreuzungspunkte sowie die entsprechenden daraus rekonstruierten Alignments in einem einfachen textbasierten Format. Verwenden Sie das gleiche Format und verifizieren Sie, dass Ihr Programm die gleichen Alignments berechnet, indem Sie

```
./Verifyalignments.sh ./linear-space.x sequences.txt | diff - alignments.txt
```

aufrufen. Sie müssen ggf. noch das zweite Argument des Shell-Skriptes (den Pfad Ihres Programms) beim obigen Aufruf anpassen.

Aufgabe 11.2 Sei T eine Sequenz der Länge n über dem Alphabet $\mathcal{A}$ und S eine *Multimenge*, d.h. eine Menge, in der Elemente mehrfach vorkommen können. Man definiert eine Multimenge als Abbildung $S : \mathcal{A} \to \mathbb{N}$ von Zeichen aus $\mathcal{A}$ auf natürliche Zahlen. Diese geben an, wie oft ein Zeichen in der Multimenge vorkommt. So ist die Multimenge $\{a, a, t, c\}$ über $\mathcal{A} = \{a, c, g, t\}$ definiert durch $S(a) = 2$, $S(c) = 1$, $S(g) = 0$, $S(t) = 1$. Sei $|S|$ die Anzahl der Zeichen in der Multimenge S, d.h. $|S| := \sum_{x \in \mathcal{A}} S(x)$.

Wir betrachten das *Multimengen-Substring-Problem* für eine Multimenge S: Es besteht darin, die Startpositionen aller Substrings von T der Länge $|S|$ zu bestimmen, in denen die Zeichen aus

S genauso oft vorkommen wie in S beschrieben. Sei zum Beispiel S wie oben definiert und $T = ggatcgtacagttt$. Dann ist $taca$ der einzige Substring von T, der aus den Zeichen von S gebildet wird, und er kommt an Position 6 vor. Für S wie oben und $T = gtacctacatacta$ ergeben sich die Substrings $taca, acat, cata, atac$ und $acta$ an den Positionen 5, 6, 7, 8 und 10.

Geben Sie einen Algorithmus mit der Laufzeit $O(n)$ an, der das Multimengen-Substring-Problem löst. Eine Beschrreibung des Algorithmus als Pseudocode reicht. Eine Implementierung ist nicht notwendig.

Aufgabe 11.3 Methoden zur effizienten Suche von signifikanten lokalen Alignments in zwei Sequenzen basieren häufig auf einem Seed-Extend Ansatz. Seeds sind dabei häufig q-grams, die in beiden Sequenzen vorkommen. Zum Aufzählen der Positionen, an denen ein q-gram in einer Sequenz vorkommt, kann man einen q-gram Index verwenden.

Implementieren Sie eine Datenstruktur bzw. Klasse $QGramIndex$, die für alle $y \in \mathcal{A}^q$ und eine gegebene Sequenz w über dem Alphabet $\mathcal{A}$ alle Positionen speichert, an denen y in w vorkommt. Die Positionen sollen 0-basiert sein und für jedes q-gram aufsteigend sortiert vorliegen. Der Zugriff auf die Positionen erfolgt durch den Integer-Code des q-Wortes. Verwenden Sie den Algorithmus zur Berechnung der Repräsentation von h_w in einem Array H, den Sie im Kontext des Fasta-Algorithmus kennengelernt haben (siehe Skript Abschnitt 4.1). Zum Generieren der Integer-Codes für q-grams können Sie Programmcode aus einem vorherigen Übungsblatt wieder verwenden.

Die Datenstruktur bzw. Klasse soll folgende Methoden zur Verfügung stellen:

- die Methode `qgram_index_new(w,q)` erzeugt für einen gegebenen String w und q die oben beschriebene Datenstruktur. Das Alphabet ist die Menge der entsprechend des ASCII-Codes sortierten Zeichen in w.
- die Methode `qgram_index_count(c)` liefert für einen Integer-Code c, $0 \leq c \leq r^q - 1$ die Anzahl der Vorkommen des q-grams y mit Integer-Code $\overline{y} = c$ in w. Dabei ist r die Alphabetgröße.
- die Methode `gram_index_get(c,i)` liefert die i-te Position in der Liste aller Positionen, an denen das q-grams y mit Integer-Code c in w vorkommt.
  Dabei muss $0 \leq i <$ `qgram_index_count(c)` gelten.
- die Methode `qgram_index_delete` gibt den Speicher für die Datenstruktur bzw. die Instanzen der Klasse wieder frei. Für Programmiersprachen mit Garbage-Collection kann dies die leere Methode sein.

Falls Sie eine Klasse implementieren, kann der Präfix `qgram_index` der obigen Methode wegfallen.

Schreiben Sie zum Testen Ihrer Implementierung ein Programm, das eine Fasta-Datei einliest und aus der ersten Sequenz einen *QGramIndex* für $q = 4$ konstruiert. Anschließend soll für alle q-grams y, die mindestens einmal vorkommen, in einer eigenen Zeile der entsprechende Integer-Code und die Liste der Positionen, an denen y in der ersten Sequenz vorkommt (durch Tabulatoren getrennt) ausgegeben werden. Das Programm erhält als Argument den Namen der Fasta-Datei sowie den Wert von q. In Stine finden Sie eine Fasta-Datei `sample.fna` zum Testen sowie eine Datei `sample-result.txt` mit dem erwarteten Ergebnis. Falls Ihr Programm `qgidx` heißt, dann sollte der Aufruf `./qgidx sample.fna 4 | diff - sample-result.txt` keine Unterschiede anzeigen.

Bitte die Lösungen zu diesen Aufgaben bis zum 16.01.2017 abgeben. Die Besprechung der

Lösungen erfolgt am 17.01.2017.

Lösung zu Aufgabe 11.1:

Die Idee ist, nacheinander, für alle j, $0 \leq j \leq n-1$, die Paare von Kreuzungspunkten $C[j]$ und $C[j+1]$ zu betrachten. Hierfür gilt nach Konstruktion $C[j] \leq C[j+1]$ und es muss ein optimales Alignment von $u[C[j]+1 \ldots C[j+1]]$ und $v[j+1]$ gebildet werden. Beachte, dass hier Strings mit 1 beginnend indiziert sind, im Programmcode aber mit 0 beginnend. Es gibt nun zwei Fälle:

Fall 1: $C[j] = C[j+1]$. Dann ist $u[C[j]+1 \ldots C[j+1]] = \varepsilon$ und das einzige Alignment von ε und $v[j+1]$ besteht nur aus der Einfügeoperation $\varepsilon \rightarrow v[j+1]$. Diese wird an das bisher konstruierte Alignment angehängt. Das Alignment wird also von links nach rechts konstruiert.

Fall 2: $C[j] < C[j+1]$. Dann ist $u[C[j]+1 \ldots C[j+1]] \neq \varepsilon$. Ein optimales Alignment von $u[C[j]+1 \ldots C[j+1]]$ und $v[j+1]$ enthält nur Ersetzungsoperationen und Löschoperationen, aber keine Einfügeoperationen, denn diese würden eine weitere Löschoperation erfordern, was die Kosten gegenüber einer direkten Ersetzung um 1 erhöhen würde. Da wir nicht wissen, ob das optimale Teilalignment eine Match-Operation enthält, wird die letzte Position eines Vorkommen von $v[j+1]$ in $u[C[j]+1 \ldots C[j+1]]$ gesucht (erster for-loop). Falls eine solche Position existiert, speichert `replacement_row` diese Position. Falls Sie nicht existiert ist speichert `replacment_row` die letzte Position des Substrings von $u[C[j]+1 \ldots C[j+1]]$. In einem zweiten for-loop wird für `replacement_row` ein eine Replacement-Optionen erzeugt und alle anderen Zeichen in $u[C[j]+1 \ldots C[j+1]]$ werden gelöscht.

Die Funktion `reconstructalignmentfromCtab` liefert die Länge des Alignments als **`return`**-Wert.

Letzter Teil der Lösung. Der Rest des Codes findet sich auf vorherigen Übungsblättern.

```
static unsigned long reconstructalignmentfromCtab(unsigned char *ali_ul,
                                                  unsigned char *ali_ll,
                                                  const unsigned long *Ctab,
                                                  const unsigned char *useq,
                                                  const unsigned char *vseq,
                                                  unsigned long vlen)
{
  unsigned long jdx, alilen = 0;
  const char gapsymbol = '-';

  for (jdx = 0; jdx < vlen; jdx++)
  {
    unsigned char b = vseq[jdx];
    /* construct an alingment of u' = u[Ctab[jdx]..C[jdx+1]-1] */
    if (Ctab[jdx] == Ctab[jdx+1])  /* u' is empty */
    {
      ali_ul[alilen] = gapsymbol;
      ali_ll[alilen++] = b; /* insertion of b */
    } else
    {
      unsigned long rowindex,
                    replacement_row = Ctab[jdx+1] - 1;

      assert(Ctab[jdx] < Ctab[jdx+1]);
      /* u' is not empty:find last position in u' at which b matches */
      for (rowindex = Ctab[jdx]; rowindex < Ctab[jdx+1]; rowindex++)
      {
```

```
        if (b == useq[rowindex])
        {
          replacement_row = rowindex;
        }
      }
      for (rowindex = Ctab[jdx]; rowindex < Ctab[jdx+1]; rowindex++)
      {
        ali_ul[alilen] = useq[rowindex];
        if (replacement_row == rowindex) /* this is the match or mismatch pos */
        {
          ali_ll[alilen++] = b;
        } else
        {
          ali_ll[alilen++] = gapsymbol; /* other symbols in u' are deleted */
        }
      }
    }
  }
  return alilen;
}

unsigned long linearspace_alignment(const unsigned char *useq,
                                    const unsigned char *vseq,
                                    unsigned long ulen,
                                    unsigned long vlen,
                                    unsigned long *Ctab,
                                    unsigned char *ali_ul,
                                    unsigned char *ali_ll,
                                    unsigned long *alilen)
{
  unsigned long distance, *EDtabcolumn, *Rtabcolumn;

  assert(ulen > 0 && vlen > 1 && Ctab != NULL);
  EDtabcolumn = malloc((ulen+1) * sizeof *EDtabcolumn);
  assert(EDtabcolumn != NULL);
  Rtabcolumn = malloc((ulen+1) * sizeof *Rtabcolumn);
  assert(Rtabcolumn != NULL);
  Ctab[0] = 0;
  Ctab[vlen] = ulen;
  evaluatecrosspoints(useq, vseq, ulen, vlen, EDtabcolumn, Rtabcolumn, Ctab, 0);
  distance = EDtabcolumn[ulen];
  free(EDtabcolumn);
  free(Rtabcolumn);
  if (ali_ul != NULL)
  {
    unsigned long alignment_costs;
    *alilen = reconstructalignmentfromCtab(ali_ul, ali_ll, Ctab, useq, vseq,
                                           vlen);
    alignment_costs = evaluate_alignment_costs (ali_ul,ali_ll,*alilen);
    assert(alignment_costs == distance);
  }
  return distance;
}
```

Lösung zu Aufgabe 11.2:

```
/* Solution to Multiset-Matching exercise. Stefan Kurtz 11.12.2013 */
```

```c
#include <stdbool.h>
#include <limits.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>

typedef struct
{
  unsigned long size;
  struct
  {
    unsigned char symbol;
    unsigned long occ;
  } *entries;
} CharMultiset;

/* brute force determining the number of matching characters, i.e.
   the number of symbols for which the values in distribution are identical
   to the corresponding values in charmultiset. The function ist used for
   testing purpose only. */

static unsigned long determine_matching_chars(const CharMultiset *charmultiset,
                                              const unsigned long *distribution)
{
  unsigned long idx, char_count = 0;

  for (idx = 0; idx < charmultiset->size; idx++)
  {
    if (distribution[charmultiset->entries[idx].symbol] ==
        charmultiset->entries[idx].occ)
    {
      char_count++;
    }
  }
  return char_count;
}

/* brute force determining if the distribution maintained in a window
   is correct, i.e. identical to a distribution computed brute force.
   Also check if number of matching_chars is identical to
   the value computed by brute force method. The function is used for
   testing purpose only. */

static void check_consistency(const unsigned long *distribution,
                              const unsigned char *window,
                              unsigned long width,
                              const CharMultiset *charmultiset,
                              unsigned long matching_chars)
{
  unsigned long idx, matching_chars2, localdist[UCHAR_MAX+1] = {0};

  for (idx = 0; idx < width; idx++)
  {
    localdist[window[idx]]++;
```

```
  }
  for (idx = 0; idx <= UCHAR_MAX; idx++)
  {
    if (distribution[idx] != localdist[idx])
    {
      fprintf(stderr,"window=%*.*s",(int) width,(int) width,window);
      fprintf(stderr,", dist[%c] = %lu != %lu = localdist[%c]\n",
                     (char) idx,distribution[idx],
                     localdist[idx],(char) idx);
      exit(EXIT_FAILURE);
    }
  }
  matching_chars2 = determine_matching_chars(charmultiset,distribution);
  if (matching_chars != matching_chars2)
  {
    fprintf(stderr,"window=%*.*s",(int) width,(int) width,window);
    fprintf (stderr,", matching_chars = %lu != %lu = matching_chars2\n",
             matching_chars,matching_chars2);
    exit(EXIT_FAILURE);
  }
}

/* initialize character distribution and alphabet for give sequence */

static unsigned long init_distribution(unsigned long *distribution,
                                       unsigned char *symbols,
                                       const char *sequence,
                                       unsigned long len)
{
  unsigned long idx, alpha_size = 0;

  for (idx = 0; idx < len; idx++)
  {
    if (distribution[(int) sequence[idx]] == 0)
    {
      symbols[alpha_size++] = sequence[idx];
    }
    distribution[(int) sequence[idx]]++;
  }
  return alpha_size;
}

/* check if value in distribution is identical in charmultiset record for
   character cc. Requires translation table mapping characters to indices
   in charmultiset. */

static bool charmultiset_occ(const unsigned long *translate,
                             const CharMultiset *charmultiset,
                             unsigned char cc,
                             unsigned long *distribution)
{
  unsigned long charidx = translate[(int) cc];

  if (charidx == ULONG_MAX)
  {
    return false;
```

```
  }
  return charmultiset->entries[charidx].occ == distribution[cc] ? true : false;
}

/* Either incremen or decrement distribution for character cc. Return
   -1 if matching_chars becomes smaller, +1 if matching_chars becomes
   larger and otherwise 0 */

static long matching_chars_change(bool increment,
                                  const unsigned long *translate,
                                  const CharMultiset *charmultiset,
                                  unsigned char cc,
                                  unsigned long *distribution)
{
  long change = 0;

  /* Note that only one of the two calls to charmultiset_occ can be true,
     as in all cases the distribution changes, but not the charmultiset. */
  if (charmultiset_occ(translate,charmultiset,cc,distribution))
  {
    change = -1;
  }
  if (increment)
  {
    distribution[(int) cc]++;
  } else
  {
    distribution[(int) cc]--;
  }
  if (charmultiset_occ(translate,charmultiset,cc,distribution))
  {
    assert(change == 0); /* as first case could not have been true at the
                            same time for the same character, due to the
                            update of the distibution. */
    change = 1; /* after update of distribution, cc occurs as many in
                   current window as in charmultiset */
  }
  return change;
}

/* This is the algorithm:
   sweep window of size multiset_size over sequence. Once index >=
   multiset_size is reached, subtract character at the left end of
   window, which falls out of it.  For any character compute number
   of matching_chars and update distribution, both in constant time.
   If number of matching characters is the same as the number of
   characters in the multiseq, then a match of length multiset_size
   is found. */

static void do_multiset_matching(const CharMultiset *charmultiset,
                                 unsigned char *sequence,
                                 unsigned long seqlen,
                                 bool check)
{
  unsigned long idx,
                multiset_size = 0,
```

```
              matching_chars = 0,
              distribution[UCHAR_MAX+1] = {0},
              translate[UCHAR_MAX+1];

for (idx = 0; idx <= UCHAR_MAX; idx++)
{
  translate[idx] = ULONG_MAX;
}
for (idx = 0; idx < charmultiset->size; idx++)
{
  multiset_size += charmultiset->entries[idx].occ;
  translate[charmultiset->entries[idx].symbol] = idx;
}
if (multiset_size > seqlen) /* sequence is too short => no solution */
{
  return;
}
for (idx = 0; idx < seqlen; idx++)
{
  long change;
  bool increment;
  if (idx >= multiset_size)
  {
    const char leftchar = sequence[idx - multiset_size];

    assert(distribution[(int) leftchar] > 0);
    if (check)
    {
      check_consistency(distribution,
                        sequence + idx - multiset_size,
                        multiset_size,
                        charmultiset,
                        matching_chars);
    }
    if (matching_chars == charmultiset->size)
    {
      printf("%lu %lu\n",idx - multiset_size,multiset_size);
    }
    increment = false; /* as leftchar is removed from distribution */
    change = matching_chars_change(increment,
                                   translate,
                                   charmultiset,
                                   leftchar,
                                   distribution);
    assert(change >= 0 || matching_chars > 0);
    matching_chars += change;
  }
  increment = true; /* as new character is added to distribution */
  change = matching_chars_change(true,
                                 translate,
                                 charmultiset,
                                 sequence[idx],
                                 distribution);
  assert(change >= 0 || matching_chars > 0);
  matching_chars += change;
}
```

```
  if (check)
  {
    check_consistency(distribution,
                      sequence + seqlen - multiset_size,
                      multiset_size,
                      charmultiset,
                      matching_chars);
  }
  if (matching_chars == charmultiset->size)
  {
    printf("%lu %lu\n",seqlen - multiset_size,multiset_size);
  }
}

/* the rest is for testing purposes */

const unsigned long random_ulong(unsigned long maxvalue)
{
  return 1 + (unsigned long) (drand48() * maxvalue);
}

/* generate random charmultisets */

static CharMultiset *charmultiset_new_random(unsigned long alpha_size,
                                             const unsigned char *symbols)
{
  unsigned long idx;
  CharMultiset *charmultiset = malloc(sizeof *charmultiset);

  assert(charmultiset != NULL);
  charmultiset->size = random_ulong(alpha_size);
  charmultiset->entries = malloc(sizeof *charmultiset->entries
                                 * charmultiset->size);
  assert(charmultiset->entries != NULL && charmultiset->size <= alpha_size);
  for (idx = 0; idx < charmultiset->size; idx++)
  {
    charmultiset->entries[idx].occ = random_ulong(3UL);
    charmultiset->entries[idx].symbol = symbols[idx];
  }
  return charmultiset;
}

/* delete it. */

static void charmultiset_delete(CharMultiset *charmultiset)
{
  if (charmultiset != NULL)
  {
    free(charmultiset->entries);
    free(charmultiset);
  }
}

/* delete it. */

void charmultiset_show(const CharMultiset *charmultiset)
```

```
{
  unsigned long idx;

  for (idx = 0; idx < charmultiset->size; idx++)
  {
    printf("%c->%lu%c",charmultiset->entries[idx].symbol,
                       charmultiset->entries[idx].occ,
                       idx < charmultiset->size - 1 ? ',' : '\n');
  }
}

static void usage(const char *progname)
{
  fprintf(stderr,"Usage: %s [check] <trials> <sequence>\n",progname);
  exit(EXIT_FAILURE);
}

int main(int argc,char *argv[])
{
  long t, trials;
  int trialarg = 0;
  bool check;
  unsigned long len, alpha_size, distribution[UCHAR_MAX+1] = {0};
  char *sequence;
  unsigned char symbols[UCHAR_MAX+1];

  if (argc == 3)
  {
    check = false;
    trialarg = 1;
  } else
  {
    if (argc == 4 && strcmp(argv[1],"check") == 0)
    {
      check = true;
      trialarg = 2;
    } else
    {
      usage(argv[0]);
    }
  }
  if (sscanf(argv[trialarg],"%ld",&trials) != 1 || trials <= 0)
  {
    usage(argv[0]);
  }
  sequence = argv[trialarg+1];
  len = strlen(sequence);
  alpha_size = init_distribution(distribution, symbols, sequence, len);
  srand48(42349421);
  for (t = 0; t < trials; t++)
  {
    CharMultiset *charmultiset = charmultiset_new_random(alpha_size,symbols);

    printf("trial %ld\n",t);
    charmultiset_show(charmultiset);
    do_multiset_matching(charmultiset,
```

```
                          (unsigned char *) sequence,
                          len,
                          check);
    charmultiset_delete(charmultiset);
  }
  exit(EXIT_SUCCESS);
}
```

Lösung zu Aufgabe 11.3:

```
#!/usr/bin/env ruby
# contains the class to store a q-word index of a sequence, testcode if file is
# executed creates such an index and prints it to stdout

# a little trick to allow relative paths from where this file is stored
# $:.unshift(File.join(File.dirname(__FILE__), '.'))
require_relative '../Qgram_code/qgram.rb'
require_relative './fasta.rb'
require "set"

class Qwidx
  def initialize(seq, q)
    @q = q
    @alphabet = seq.downcase.split(//).uniq.sort
    alphasize = @alphabet.length
    if seq.length < q
      raise ArgumentError, "q=#{q} too large, must not be larger than " +
                           "#{seq.length}=length of sequence"
    end
    @num_qgrams = alphasize**q
    count = Array.new(@num_qgrams) {0}
    @partialsums = Array.new(@num_qgrams)
    qgram = Qgram.new(q, @alphabet)
    qgram.each_code(seq) do |code|
      count[code] += 1
    end
    @partialsums[0] = 0
    1.upto(@num_qgrams-1).each do |code|
      @partialsums[code] = @partialsums[code-1] + count[code-1]
    end
    @posarray = Array.new(seq.length - q + 1)
    qgram.each_code_with_pos(seq) do |code, pos|
      @posarray[@partialsums[code]] = pos
      @partialsums[code] += 1
    end
  end

  def num_qgrams_get()
    return @num_qgrams
  end

  def alphabet_get()
    return @alphabet
  end

  def q_get()
    return @q
```

```
  end

  def count(code)
    if code == 0
      return @partialsums[code]
    elsif code < @num_qgrams
      return @partialsums[code] - @partialsums[code-1]
    else
      raise ArgumentError, "code=#{code} too large, must be smaller than " +
                           "#{@num_qgrams}=number of q-grams"
    end
  end

  def get(code,idx)
    numberofpositions = self.count(code)
    if code == 0
      offset = 0
    else
      offset = @partialsums[code-1]
    end
    if idx < 0 or idx >= numberofpositions
      raise ArgumentError, "idx=#{idx} too large, must be smaller than " +
                           "#{numberofpositions}=number of positions for " +
                           "code #{code}"
    end
    return @posarray[offset + idx]
  end

  def delete()
   return
  end
end

def checkposlist(seq,q,qgram,list)
  code = list[0]
  list.shift
  list.each do |pos|
    cc = qgram.encode(seq[pos..pos+q-1])
    if cc != code
      STDERR.puts "#{$0}: cc = #{cc} != #{code} = code not expected"
      exit 1
    end
  end
end

def qwidx_verify(qwidx,seq)
  alphabet = qwidx.alphabet_get()
  q = qwidx.q_get()
  qgram = Qgram.new(q, alphabet)
  list = Array.new()
  pos_set = Set.new()
  0.upto(qwidx.num_qgrams_get()-1).each do |code|
    numberofpositions = qwidx.count(code)
    if numberofpositions > 0
      list.clear()
      list.push(code)
```

```
      0.upto(numberofpositions-1).each do |idx|
        pos = qwidx.get(code,idx)
        list.push(pos)
        if pos_set.member?(pos)
          STDERR.puts "#{$0}: pos #{pos} already occurs in pos_set"
          exit 1
        end
        if pos < 0 or pos > seq.length - q + 1
          STDERR.puts "#{$0}: pos #{pos} too large"
          exit 1
        end
        pos_set.add(pos)
      end
      puts list.join("\t")
      checkposlist(seq,q,qgram,list)
    end
  end
  if pos_set.size() != seq.length - q + 1
    STDERR.puts "#{$0}: pos_set has incorrect size"
    exit 1
  end
end

if $0 == __FILE__
  usage = "USAGE: #$0 fasta-file q\n" \
    "where q is the length of the q-grams"
  if ARGV.length != 2
    STDERR.puts usage
    exit 1
  end
  q = 0
  head = seq = nil
  begin
    head, seq = Fasta::read(ARGV[0])
  rescue => e
    STDERR.puts "reading of #{ARGV[0]} error: #{e.message}"
    STDERR.puts usage
    exit 1
  end
  begin
    q = Integer(ARGV[1])
    raise if q < 1
  rescue
    STDERR.puts "#{ARGV[1]} could not be parsed as an integer > 0"
    STDERR.puts usage
    exit 1
  end
  qwidx = Qwidx.new(seq,q)
  qwidx_verify(qwidx,seq)
end
```