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

Grundlagen der Sequenzanalyse
Wintersemester 2016/2017
Übungen zur Vorlesung: Ausgabe am 06.12.2016

Punkteverteilung:

Aufgabe 8.1: 6 Punkte, Aufgabe 8.2: 4 Punkte,

Abgabe bis zum 12.12.

Aufgabe 8.1 Implementieren Sie in Ruby oder in C ein Programm zur Berechnung der Koordinaten optimaler lokaler Alignments mit Hilfe des Smith-Waterman-Algorithmus. Ihr Programm soll folgendermaßen aufgerufen werden:

```
swalign.<suffix> indelscore scorematrixfile inputfile
```

Dabei ist

- `suffix` der von der verwendeten Programmiersprache abhängige Suffix des ausführbaren Programms, also z.B. `x` für das ausführbare (also compilierte und durch den Linker prozessierte) C-Programm oder `rb` für ein Ruby-Skript.
- `indelscore` ist der Score für Einfügungen und Löschungen, in diesem Kontext ein negativer Integer.
- `scorematrixfile` ist der Dateiname für die Similarity-Scorematrix im Format des Programms Blast.
- `inputfile` ist der Name einer Datei, die mindestens zwei Zeilen enthält. Jede einzelne Zeile wird als Sequenz betrachtet und Ihr Programm soll alle Paare gegen alle anderen vergleichen. Falls die Datei n Zeilen, also n-Sequenzen $s_0, \ldots, s_{n-1}$ enthält, dann sollen alle Paare s_i, s_j mit $0 \leq i < j \leq n-1$ verglichen werden.

Damit Sie sich auf das Wesentliche konzentrieren können, finden Sie im Material zu dieser Aufgabe einen Testrahmen (in C und Ruby), Testdaten (Proteinsequenzen) und die erwarteten Ergebnisse:

Hinweise zum Testrahmen in Ruby (siehe `ruby-solution`)

- `all_against_all.rb` enthält einen Iterator, der für eine gegebene Sequenzdatei alle oben definierten Sequenzpaare aufzählt.
- `scorematrix.rb` enthält eine Klasse zum Einlesen einer Scorematrix.
- `swalign.rb` liest die Scorematrix ein, deklariert für Ihre Anwendung relevante Strukturen, zerlegt das `ARGV`-Array, nutzt den Iterator zum Aufzählen der Sequenzpaare und liefert die Ergebnisse im passenden Format. Sie selbst müssen in dieser Datei nur die Methode `smithwaterman_algorithm` implementieren. Die entsprechende zu füllende Struktur und die **`return`**-Anweisung sind bereits eingefügt. Die Struktur speichert 5 Werte, nämlich (in dieser Reihenfolge):

1. die Startposition des alignierten Substrings in der Sequenz u
2. die Länge des alignierten Substrings in u,
3. die Startposition des alignierten Substrings in der Sequenz v
4. die Länge des alignierten Substrings in v,
5. den Score des optimalen lokalen Alignments

Damit Ihre Lösung exakt der erwartenden Lösung entspricht, müssen Sie für den Fall, dass es in der Matrix L_σ mehrere Einträge mit dem gleichen maximalen Score-Wert gibt, bestimmte Auswahlregeln beachten. Diese sind in der Datei `swalign.h` beschrieben.

- Durch `make test-gsa` wird ein minimaler Test mit den beiden Sequenzen aus dem GSA-Script durchgeführt, die Sie in der Datei `gsa-seqpair.txt` finden. Die verwendete Scorematrix steht in `unitscore.txt`. Das erwartete Ergebnis steht in `gsa-align.txt`. Wenden Sie Ihr Programm zunächst auf diesen Testfall an. Die Suche nach Fehlern wird erleichtert, da die Berechnung des Ergebnisses im GSA-Skript detailliert dokumentiert ist.
- Durch `make test-small` wird ein Test mit 18 Sequenzen aus der Datei `seq18.txt`, dem Indelscore -4 und der Scorematrix aus der Datei `blosum62.txt` durchgeführt. Das erwartete Ergebnis finden Sie in der Datei `align18.txt`.

Hinweise zum Testrahmen in C:

Dieser Testrahmen besteht aus 6 C-Dateien und 5 Header Dateien, deren Verständnis für diese Aufgabe nicht wichtig ist.

Zum Compilieren der C-Sourcen verwenden Sie `make`. Sie müssen lediglich in der Datei `swalign.c` eine Funktion

```
void smithwaterman_algorithm(OptimalLocalAlignmentCoords *coords,
                             const Scorematrix *scorematrix,
                             Score insertionscore,
                             Score deletionscore,
                             const unsigned char *useq,
                             unsigned long ulen,
                             const unsigned char *vseq,
                             unsigned long vlen)
```

implementieren. Diese soll die durch `useq` und `vseq` referenzierten Sequenzen der Längen `ulen` und `vlen` alignieren. Dabei sind die Scores durch die Parameter `scorematrix`, `insertionscore` und `deletionscore` bestimmt. Sie müssen die Funktion

```
Score scorematrix_score_get(const Scorematrix *scorematrix,unsigned char a,
                            unsigned char b);
```

aufrufen, um den Score für ein Zeichenpaar a und b zu ermitteln. Ein Zeichen aus den zu alignierenden Sequenzen wird Ihnen im Testrahmen als Index im Wertebereich von 0 bis 22 zur Verfügung gestellt. Dieses vereinfacht den Zugriff auf die Scorematrix.

Die Koordinaten eines optimalen lokalen Alignments werden in `coords` gespeichert. Die einzelnen Komponenten entsprechen denen in der Ruby-Struct, die oben beschrieben wurde. Damit Ihre Lösung exakt der erwarteten Lösung entspricht, müssen Sie für den Fall, dass es in L_σ mehrere

Einträge mit dem gleichen maximalen Score-Wert gibt, bestimmte Auswahlregeln beachten. Diese sind in der Datei `swalign.h` beschrieben.

Überlegen Sie, welche Informationen und Funktionen zusätzlich zu der DP-Matrix noch zur Berechnung der Koordinaten eines optimalen lokalen Alignments notwendig sind und erweitern Sie ggf. Ihren Programmcode aus früheren Übungen um die entsprechende Funktionalität. Überlegen Sie sich auch, wie Sie am einfachsten die Startpositionen der optimalen lokalen Alignments bestimmen können. Ein offensichtliche Lösung über Backtracing benötigt $O(mn)$ Platz. Es gibt auch eine Lösung ohne Backtracing, die sich in $O(m)$ Platz implementieren lässt. Falls Sie diese Lösung implementieren, können Sie zur Vereinfachung auch $O(mn)$ Platz verwenden.

Ihr Programm sollte die beiden Tests `test-gsa` und `test-small` (siehe Makefile) bestehen.

Aufgabe 8.2 Eine zentrale Funktion beim Linear-Space-Alignment-Algorithmus für zwei Sequenzen u (mit $|u| = m$) und v (mit $|v| = n$) ist die Funktion *evaluateallcolumns* (Details siehe Skript). Implementieren Sie diese Funktion und gehen Sie dabei schrittweise vor (so wie im Vorlesungsskript beschrieben), d.h.:

1. Implementieren Sie zuerst die Funktionen *firstEDtabcolumn* und *nextEDtabcolumn*.
2. Erweitern Sie diese Funktionen dann zu den Funktionen *firstEDtabRtabcolumn* und *nextEDtabRtabcolumn*.
3. Kombinieren Sie diese Funktionen zu der gewünschten Funktion *evaluateallcolumns*.

Stellen Sie sicher, dass nie mehr als $O(m)$ Speicherplatz verwendet wird und keine (Sub-)Strings in mehrfacher Kopie erzeugt werden. Idealerweise allozieren Sie einmal zwei Arrays der Länge $m + 1$ für E und R und überschreiben diese im Laufe des Algorithmus. Eine konstante Anzahl von Variablen kann dabei zur temporären Speicherung von Zwischenwerten verwendet werden.

Bitte die Lösungen zu diesen Aufgaben bis zum 12.12.2016 abgeben. Die Besprechung der Lösungen erfolgt am 13.12.2016.

Lösung zu Aufgabe 8.1:

Eine Lösung in C in linearem Platz ohne Backtracing

```
/*
  Copyright (c) 2016 Stefan Kurtz <kurtz@zbh.uni-hamburg.de>
  Copyright (c) 2016 Center for Bioinformatics, University of Hamburg

  Permission to use, copy, modify, and distribute this software for any
  purpose with or without fee is hereby granted, provided that the above
  copyright notice and this permission notice appear in all copies.

  THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
  WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
  MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
  ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
```

```
  WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
  ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
  OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
*/

#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "scorematrix.h"
#include "swalign.h"

typedef struct
{
  unsigned long aligned_u, aligned_v;
  Score score;
} LMatrixEntry;

typedef struct
{
  unsigned long start_row, start_column, aligned_u, aligned_v;
  Score score;
} BestCoordinate;

static void sw_eval_columns(BestCoordinate *maxscore_entry,
                            const Scorematrix *scorematrix,
                            Score insertionscore,
                            Score deletionscore,
                            const unsigned char *useq,
                            unsigned long ulen,
                            const unsigned char *vseq,
                            unsigned long vlen)
{
  unsigned long idx;
  const unsigned char *uptr, *vptr;
  LMatrixEntry *scol = malloc(sizeof *scol * (ulen + 1));

  assert(maxscore_entry != NULL);
  for (idx = 0; idx <= ulen; idx++)
  {
    scol[idx].score = 0;
    scol[idx].aligned_u = 0;
    scol[idx].aligned_v = 0;
  }
  maxscore_entry->score = 0;
  maxscore_entry->start_row = 0;
  maxscore_entry->start_column = 0;
  maxscore_entry->aligned_u = 0;
  maxscore_entry->aligned_v = 0;
  for (vptr = vseq; vptr < vseq + vlen; vptr++)
  {
    LMatrixEntry nw = *scol, *scolptr;
    const unsigned long column = (unsigned long) (vptr - vseq + 1);

    for (scolptr = scol+1, uptr = useq; uptr < useq + ulen; scolptr++, uptr++)
    {
```

```
#define MINIMIZE_BRANCHING
#ifdef MINIMIZE_BRANCHING
      LMatrixEntry valnorth, valwest;
#else
      Score val;
#endif
      const LMatrixEntry we = *scolptr;

      scolptr->score = nw.score +
                       scorematrix_score_get(scorematrix,*uptr,*vptr);
      scolptr->aligned_u = nw.aligned_u + 1;
      scolptr->aligned_v = nw.aligned_v + 1;
#ifdef MINIMIZE_BRANCHING
      valnorth.score = (scolptr-1)->score + deletionscore;
      valnorth.aligned_u = (scolptr-1)->aligned_u + 1;
      valnorth.aligned_v = (scolptr-1)->aligned_v;
      valwest.score = we.score + insertionscore;
      valwest.aligned_u = we.aligned_u;
      valwest.aligned_v = we.aligned_v + 1;
#define FIRSTWORSETHANSECOND(VAL1,VAL2)\
                  (((VAL1)->score > (VAL2)->score ||\
                  ((VAL1)->score == (VAL2)->score &&\
                  (VAL1)->aligned_u + (VAL1)->aligned_v >= \
                  (VAL2)->aligned_u + (VAL2)->aligned_v)) ? false : true)
      if (FIRSTWORSETHANSECOND(scolptr,&valnorth))
      {
        *scolptr = valnorth;
      }
      if (FIRSTWORSETHANSECOND(scolptr,&valwest))
      {
        *scolptr = valwest;
      }
#else
      if ((val = (scolptr-1)->score + deletionscore) >= scolptr->score)
      {
        if (val > scolptr->score)
        {
          scolptr->score = val;
          scolptr->aligned_u = (scolptr-1)->aligned_u + 1;
          scolptr->aligned_v = (scolptr-1)->aligned_v;
        } else
        {
          if (scolptr->aligned_u + scolptr->aligned_v <
              (scolptr-1)->aligned_u + 1 + (scolptr-1)->aligned_v)
          {
            scolptr->aligned_u = (scolptr-1)->aligned_u + 1;
            scolptr->aligned_v = (scolptr-1)->aligned_v;
          }
        }
      }
      if ((val = we.score + insertionscore) >= scolptr->score)
      {
        if (val > scolptr->score)
        {
          scolptr->score = val;
          scolptr->aligned_u = we.aligned_u;
```

```
          scolptr->aligned_v = we.aligned_v + 1;
        } else
        {
          if (scolptr->aligned_u + scolptr->aligned_v <
              we.aligned_u + we.aligned_v + 1)
          {
            scolptr->aligned_u = we.aligned_u;
            scolptr->aligned_v = we.aligned_v + 1;
          }
        }
      }
#endif
      if (scolptr->score > 0)
      {
        if (maxscore_entry->score < scolptr->score ||
            (maxscore_entry->score == scolptr->score &&
             scolptr->aligned_u + scolptr->aligned_v >
                maxscore_entry->aligned_u + maxscore_entry->aligned_v))
        {
          maxscore_entry->score = scolptr->score;
          maxscore_entry->start_row
            = (unsigned long) (scolptr - scol) - scolptr->aligned_u;
          maxscore_entry->start_column = column - scolptr->aligned_v;
          maxscore_entry->aligned_u = scolptr->aligned_u;
          maxscore_entry->aligned_v = scolptr->aligned_v;
        }
      } else
      {
        scolptr->score = 0;
        scolptr->aligned_u = 0;
        scolptr->aligned_v = 0;
      }
      nw = we;
    }
  }
  free(scol);
}

void smithwaterman_algorithm(OptimalLocalAlignmentCoords *coords,
                             const Scorematrix *scorematrix,
                             Score insertionscore,
                             Score deletionscore,
                             const unsigned char *useq,
                             unsigned long ulen,
                             const unsigned char *vseq,
                             unsigned long vlen)
{
  BestCoordinate maxscoreentry;
  const bool keeporder = ulen < vlen ? true : false;

  sw_eval_columns(&maxscoreentry,
                  scorematrix,
                  insertionscore,
                  deletionscore,
                  keeporder ? useq : vseq,
                  keeporder ? ulen : vlen,
```

```
                  keeporder ? vseq : useq,
                  keeporder ? vlen : ulen);
  if (maxscoreentry.score > 0)
  {
    if (keeporder)
    {
      coords->ustart = maxscoreentry.start_row;
      coords->usubstringlength = maxscoreentry.aligned_u;
      coords->vstart = maxscoreentry.start_column;
      coords->vsubstringlength = maxscoreentry.aligned_v;
    } else
    {
      coords->vstart = maxscoreentry.start_row;
      coords->vsubstringlength = maxscoreentry.aligned_u;
      coords->ustart = maxscoreentry.start_column;
      coords->usubstringlength = maxscoreentry.aligned_v;
    }
    coords->score = maxscoreentry.score;
  } else
  {
    coords->ustart = 0;
    coords->usubstringlength = 0;
    coords->vstart = 0;
    coords->vsubstringlength = 0;
    coords->score = 0;
  }
}

void smithwaterman_algorithm_void(void *data,
                                  const unsigned char *useq,
                                  unsigned long ulen,
                                  const unsigned char *vseq,
                                  unsigned long vlen)
{
  SwalignScoresAndResult *sar = (SwalignScoresAndResult *) data;

  smithwaterman_algorithm(&sar->coords,
                          sar->scorematrix,
                          sar->insertionscore,
                          sar->deletionscore,
                          useq,
                          ulen,
                          vseq,
                          vlen);
}
```

Eine Lösung in Ruby in quadratischem Platz ohne Backtracing

Die Idee dieser Lösung ist, mit jedem Score-Wert in der DP-Matrix noch die Start-Koordinaten des entsprechenden lokalen Alignments zu speichern. D.h. immer dann, wenn ein Eintrag den Score 0 hat, beginnt ein lokales Alignment. Diese Koordinaten werden entlang der maximierenden Kanten vererbt. Um eindeutige Lösungen zu bekommen, bevorzugt man Ersetzungskanten vor Löschkan-

ten und Löschkanten vor Einfügekanten, für den Fall, dass es mehrere eingehende maximierende Kanten gibt.

```
#!/usr/bin/env ruby

require_relative '../Scorematrix_code/scorematrix.rb'
require 'all_against_all.rb'

def showdpmatrix(dptable,ulen,vlen)
  0.upto(vlen) do |j|
    0.upto(ulen) do |i|
      terminator = if i < ulen then " " else "\n" end
      print "#{dptable[i][j].score}#{terminator}"
    end
  end
end

SWentry = Struct.new("SWentry",:score, :start_u, :start_v)

OptimalLocalAlignmentCoords = Struct.new("OptimalLocalAlignmentCoords",
    :ustart, # start position of local opt. alignment in <useq>
    :usubstringlength, # length of substring of local opt. align. in <useq>
    :vstart, # start position of local opt. alignment in <vseq>
    :vsubstringlength, # length of substring of local opt. align. in <vseq> */
    :score)

def smithwaterman_algorithm(scorematrix,insertionscore,deletionscore,useq,vseq)
  ulen = useq.length
  vlen = vseq.length
  optimalAlignedCoords = OptimalLocalAlignmentCoords.new(0,0,0,0,0)
             # coordinates of substring pair with optimal loc. alignment
  maxrow = 0 # maxrow + maxcolumn are maximum sum of end coordinates seen so far
  maxcolumn = 0
  dptable = Array.new(ulen+1) { Array.new(vlen+1) { SWentry.new(0,0,0) } }
  0.upto(ulen) do |i|
    dptable[i][0].score = 0
    dptable[i][0].start_u = i
    dptable[i][0].start_v = 0
  end
  1.upto(vlen) do |j|
    dptable[0][j].score = 0
    dptable[0][j].start_u = 0
    dptable[0][j].start_v = j
    1.upto(ulen) do |i|
      nws = dptable[i-1][j-1].score +
            scorematrix.getscore(useq[i-1].chr,vseq[j-1].chr)
      ns = dptable[i-1][j].score + deletionscore
      ws = dptable[i][j-1].score + insertionscore
      dptable[i][j].score = [nws, ns, ws].max
      if dptable[i][j].score <= 0
        dptable[i][j].score = 0
        dptable[i][j].start_u = i
        dptable[i][j].start_v = j
      else
        dptable[i][j].start_u = nil
        dptable[i][j].start_v = nil
        maxval = nil
```

```
        [[nws,i-1,j-1],[ns,i-1,j],[ws,i,j-1]].each do |s,p,q|
          if s == dptable[i][j].score and (maxval.nil? or
             (maxval > dptable[p][q].start_u + dptable[p][q].start_v))
            dptable[i][j].start_u = dptable[p][q].start_u
            dptable[i][j].start_v = dptable[p][q].start_v
            maxval = dptable[i][j].start_u + dptable[i][j].start_v
          end
        end
        if optimalAlignedCoords.score < dptable[i][j].score or
           (optimalAlignedCoords.score == dptable[i][j].score and
            i - dptable[i][j].start_u + j - dptable[i][j].start_v >
              optimalAlignedCoords.usubstringlength +
              optimalAlignedCoords.vsubstringlength)
          optimalAlignedCoords.score = dptable[i][j].score
          optimalAlignedCoords.ustart = dptable[i][j].start_u
          optimalAlignedCoords.usubstringlength = i - dptable[i][j].start_u
          optimalAlignedCoords.vstart = dptable[i][j].start_v
          optimalAlignedCoords.vsubstringlength = j - dptable[i][j].start_v
        end
      end
    end
  end
  return optimalAlignedCoords
end

if ARGV.length != 3
  STDERR.puts "Usage: #{$0} indelscore scorematrixfile sequencefile"
  exit
end

indelscore = ARGV[0].to_i
scorematrixfile = ARGV[1]
scorematrix = Scorematrix.new(scorematrixfile)
sequencefile = ARGV[2]

all_against_all(sequencefile) do |i,j,useq,vseq|
  result = [i,j]
  if useq.length < vseq.length
    maxscorealign = smithwaterman_algorithm(scorematrix,indelscore,indelscore,
                                            useq,vseq)
    result.push(maxscorealign.ustart)
    result.push(maxscorealign.usubstringlength)
    result.push(maxscorealign.vstart)
    result.push(maxscorealign.vsubstringlength)
  else
    maxscorealign = smithwaterman_algorithm(scorematrix,indelscore,indelscore,
                                            vseq,useq)
    result.push(maxscorealign.vstart)
    result.push(maxscorealign.vsubstringlength)
    result.push(maxscorealign.ustart)
    result.push(maxscorealign.usubstringlength)
  end
  result.push(maxscorealign.score)
  puts result.join("\t")
end
```

Lösung zu Aufgabe 8.2:

Hier ist der erste Teil der Lösung. Der Rest des Codes folgt im Kontext einer weiteren Übungsaufgabe.

```
#define ADDUNIT(A,B) ((A) == (B) ? 0 : 1)
#define GAP          ((unsigned char) '-')
#define DIV2(N)      ((N) >> 1)

/*
   The following function computes the first column of the E and R table as
   described in the handout of the lecture on ``A Linear Space Alignmen
   Algorithm''.  EDtabcolumn takes a column of the E table and should
   contain the first column after completion of the function. Analog,
   Rtabcolumn takes a column of the R table and should contain the first column
   after completion of the function (this is the initialization!).
   ulen is the length of the first sequence.
   */
static void firstEDtabRtabcolumn(unsigned int *EDtabcolumn,
                                 unsigned int *Rtabcolumn,
                                 unsigned int ulen)
{
  unsigned int rowindex;

  for (rowindex=0; rowindex <= ulen; rowindex++) {
    EDtabcolumn[rowindex] = rowindex;
    Rtabcolumn[rowindex]  = rowindex;
  }
}

/*
   The following function computes a column of the E and R table under the
   assumption that EDtabcolumn and Rtabcolumn contain the previous columns
   of the corresponding tables. The function overwrites EDtabcolumn and
   Rtabcolumn with the new columns. b is the character of the
   sequence vseq corresponding to to column which needs to be calculated. Useq
   is the first sequence and ulen its length.

   colindex is the number of the column to be calculated and midcolumn
   is the column, where the tables are divided.
   */
static void nextEDtabRtabcolumn(unsigned int *EDtabcolumn,
                                unsigned int *Rtabcolumn,
                                unsigned int colindex,
                                unsigned int midcolumn, unsigned char b,
                                const unsigned char *useq, unsigned int ulen)
{
  unsigned int rowindex, val,
               northwestEDtabentry,
               westEDtabentry = EDtabcolumn[0], /* saves the first entry of
                                                   EDtabcolumn */
               northwestRtabentry,
               westRtabentry = 0;
  bool updateRtabcolumn = false;

  EDtabcolumn[0]++;
  if (colindex > midcolumn) {
```

```
    updateRtabcolumn = true;
  }
  for (rowindex = 1; rowindex <= ulen; rowindex++) {
    northwestEDtabentry = westEDtabentry;
    northwestRtabentry  = westRtabentry;
    westEDtabentry = EDtabcolumn[rowindex];
    westRtabentry  = Rtabcolumn[rowindex];
    EDtabcolumn[rowindex]++; /* 1. recurrence (Insertion) */
    /* Rtabcolumn[rowindex] is unchanged */
    /* 2. recurrence (Replacement): */
    val = northwestEDtabentry + ADDUNIT(useq[rowindex-1],b);
    if (val < EDtabcolumn[rowindex]) {
      EDtabcolumn[rowindex] = val;
      if (updateRtabcolumn)
        Rtabcolumn[rowindex] = northwestRtabentry;
    }
    /* 3. recurrence (Deletion): */
    val = EDtabcolumn[rowindex-1]+1;
    if (val < EDtabcolumn[rowindex]) {
      EDtabcolumn[rowindex] = val;
      if (updateRtabcolumn)
        Rtabcolumn[rowindex] = Rtabcolumn[rowindex-1];
    }
  }
}

/*
   The following function computes the rightmost EDtabcolumn and Rtabcolumn.
   midcolumn is the actual column, where the tables are divided.
   useq and vseq are the sequences with the corresponding length ulen and vlen.
 */
static unsigned int evaluateallcolumns(unsigned int *EDtabcolumn,
                                       unsigned int *Rtabcolumn,
                                       unsigned int midcol,
                                       const unsigned char *useq,
                                       const unsigned char *vseq,
                                       unsigned int ulen, unsigned int vlen)
{
  unsigned int colindex;

  firstEDtabRtabcolumn(EDtabcolumn, Rtabcolumn, ulen);

  for (colindex = 1; colindex <= vlen; colindex++) {
    nextEDtabRtabcolumn(EDtabcolumn, Rtabcolumn, colindex, midcol,
                        vseq[colindex-1], useq, ulen);
  }
  return EDtabcolumn[ulen];
}
```