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

Grundlagen der Sequenzanalyse
Wintersemester 2016/2017
Übungen zur Vorlesung: Ausgabe am 25.10.2016

Punkteverteilung: Aufgabe 2.1: 4 Punkte, Aufgabe 2.2: 3 Punkte, Aufgabe 2.3: 3 Punkte

Abgabe bis zum 31.10. Hinweis zu Aufgaben 2.1 und 2.3: Anstatt Pseudocode kann auch lauffähiger Programmcode abgegeben werden.

Aufgabe 2.1 Seien u und v zwei beliebige Sequenzen der Länge m bzw. n. Für alle $i, 0 \leq i \leq m$ und $j, 0 \leq j \leq n$ bezeichne $Aligns(i,j)$ die Anzahl der Alignments von $u[1\dots i]$ und $v[1\dots j]$. In der Vorlesung wurde die folgende Rekurrenz für $Aligns$ entwickelt:

$$Aligns(i,j) \quad = \begin{cases} 1 & \text{falls } i = 0 \text{ oder } j = 0 \\ Aligns(i-1,j)+ & \\ Aligns(i,j-1)+ & \\ Aligns(i-1,j-1) & \text{sonst} \end{cases}$$

1. Schreiben Sie in einer Programmiersprache Ihrer Wahl oder in Pseudocode ein Programm `aligns`, das direkt nach obiger Rekurrenz den Wert $Aligns(m,n)$ berechnet. D.h. schreiben Sie $Aligns$ direkt als rekursive Funktion auf. Das Programm und die Funktion sollen die Parameter m und n als (Kommandozeilen-)Argumente bekommen. Die maximalen Werte für m und n sollen jeweils 15 sein. Bei Eingaben, die größer als 15 sind, soll das Programm mit einer Fehlermeldung abbrechen. Das Programm soll eine Ausgabe in folgendem Format liefern:

   ```
   m n result (c calls)
   ```

   Dabei ist `c` die Anzahl der rekursiven Aufrufe.

   Beispiel: Der Aufruf von `aligns 4 4` soll das folgende Ergebnis liefern:

   ```
   4 4 321 (481 calls)
   ```

2. (*Nur für Lösungen, die in einer Programmiersprache abgegeben werden*)
   Berechnen Sie $Aligns(15, 11) = 921406335$. Wieviele Aufrufe der Funktion $Aligns$ benötigt Ihr Programm, um diesen Wert zu bestimmen?

3. Schreiben Sie nun ein Programm (bzw. geben Sie Pseudocode an), welches $Aligns(m,n)$ nach der obigen Rekurrenz mit Hilfe einer $(m+1)\times(n+1)$-Matrix $Alignstab$ berechnet. Dabei gilt für alle $i, j, 0 \leq i \leq m, 0 \leq j \leq n$ die Gleichung

   $$Alignstab[i,j] = Aligns(i,j)$$

   Bitte beachten Sie dabei, dass das Füllen einer Matrix nach einer Rekurrenz *nicht* zwangsweise bedeutet, dass die Matrixwerte durch rekursive Funktionsaufrufe gefüllt wird. Es ist bei dieser Teilaufgabe eine *nicht*-rekursive Lösung gefordert.

   Das Programm soll eine Ausgabe in folgendem Format liefern:

   ```
   m n result (a accesses)
   ```

   Dabei ist `a` die Anzahl der Zugriffe auf $Alignstab$.

   Beispiel: Ein Aufruf von `alignstab 3 3` soll folgendes Ergebnis liefern:

   ```
   3 3 63 (27 accesses)
   ```

4. (*Nur für Lösungen, die in einer Programmiersprache abgegeben werden*)
   Berechnen Sie $Aligns(15, 11)$ mit Ihrem zweiten Programm. Wieviele Zugriffe auf die Matrix $Alignstab$ benötigt Ihr Programm, um obigen Wert zu bestimmen?
5. Beschreiben Sie den konzeptionellen Unterschied zwischen einer echt rekursiven Implementierung einer Rekurrenz und einer Implementierung der Rekurrenz mit Hilfe einer Matrix.

Aufgabe 2.2 Im Skript zu dieser Vorlesung finden sich häufig Mengendefinitionen, wie z.B. im Kapitel 2:

$$\mathcal{A}^i = \begin{cases} \{\varepsilon\} & \text{if } i = 0 \\ \{aw \mid a \in \mathcal{A}, w \in \mathcal{A}^{i-1}\} & \text{if } i > 0 \end{cases}$$

Diese Defintion läßt sich wie folgt lesen:

Die Menge $\mathcal{A}^0$ aller Strings der Länge 0 ist $\{\varepsilon\}$, wobei ε das leere Wort bezeichnet. Die Menge $\mathcal{A}^i$ aller Strings der Länge $i > 0$ ist definiert als die Menge aller aw, wobei a ein Element aus der Menge $\mathcal{A}$ ist und w ein Element aus der Menge $\mathcal{A}^{i-1}$, also der Menge der Strings der Länge $i - 1$.

1. Beschreiben Sie wie im obigen Beispiel aus dem Skript folgende Mengendefinitionen in ganzen Sätzen:
   a) $D_k := \{2 \cdot i \mid 0 \leq i \leq k\}$
   b) $P_w := \{(i, j) \mid 1 \leq i < j \leq n, (w[i], w[j]) \in B, j - i - 1 \geq l\}$. w steht hierbei für einen String über dem Alphabet $\mathcal{A} = \{A, C, G, U\}$, B bezeichnet die Menge $\{(A, U), (U, A), (C, G), (G, C), (G, U), (U, G)\}$ und l und n sind ganze Zahlen.
2. Geben Sie zu folgenden in ganzen Sätzen definierten Mengen $H(w, v)$ und $M_k(w, v)$ formale Definitionen an:
   a) Für zwei Strings w und v gleicher Länge ist $H(w, v)$ die Anzahl aller Positionen in w und v mit unterschiedlichen Zeichen.
   b) Für zwei Strings w und v und eine positive ganze Zahl k ist $M_k(w, v)$ die Menge aller Paare von Positionen in w und v an denen ein Treffer (d.h. ein gemeinsamer Substring) der Länge k beginnt.

Aufgabe 2.3 Seien u und v zwei Sequenzen der Länge m bzw. n. Eine gemeinsame Subsequenz von u und v ist eine Folge $(i_1, j_1), \ldots, (i_r, j_r)$, so dass gilt:

1. $1 \leq i_1 < \ldots < i_r \leq m$ und
2. $1 \leq j_1 < \ldots < j_r \leq n$ und
3. $u[i_1] = v[j_1], \ldots, u[i_r] = v[j_r]$.

Bitte beachten Sie, dass jede gemeinsame Subsequenz eine Subsequenz (wie in der Vorlesung definiert) ist. Zusätzlich ist gefordert, dass die Zeichen der Sequenzen an den Positionen der Subsequenz identisch sind, siehe Bedingung 3.

Sei $lcs(u, v)$ die Länge der längsten gemeinsamen Subsequenz von u und v. Z.B. ist $lcs(acatcagac, aactacgc) = 6$, denn $(1, 1), (2, 3), (3, 5), (5, 6), (7, 7), (9, 8)$ ist die längste gemeinsame Subsequenz von $acatcagac$ und $aactacgc$. Sie repräsentiert den String $acacgc$.

Geben Sie einen rekursiven Algorithmus an, der $lcs(u, v)$ für zwei Sequenzen u und v berechnet.

Die Lösungen zu diesen Aufgaben werden am 01.11.2016 besprochen.

Lösung zu Aufgabe 2.1:

C-Variante		
	calls/accesses	Laufzeit auf Intel i5, 3.4 GHz:
rekursive Variante:	1382109502	$\approx$ 2 Sekunden
iterative Variante:	$3mn$=495	$\approx$ 0 Sekunden
Ruby-Variante		
	calls/accesses	Laufzeit auf i5, MHz:
rekursive Variante:	1382109502	$\approx$ 2003 Sekunden
iterative Variante:	$3mn$=495	$\approx$ 0.04 Sekunden

Das Konzept bei rekursiven aufrufen ist, dass jeder Teilwert zur Berechnung eines Ergebnisses oder Zwischenergebnisses einzeln und explizit berechnet wird. Bei der Tabulierten (Dynamischen Programmierung) werden alle häufig berechneten Werte abgespeichert und müssen daher nur einmal berechnet werden. Die Resultate werden dann implizit wiederverwertet.

C-Variante

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

typedef unsigned long long Matrixtype;

#define MATRIXFORMAT "%llu"

static unsigned long long countcalls = 0,
                          countaccesses = 0;

#define READINT(ARGNUM,VAR,STR)\
        if (sscanf(argv[ARGNUM],"%d",&readint) != 1 || readint < 0) {\
          fprintf(stderr,"Usage: %s mode m n\n"\
                         "%s argument must be positive integer",argv[0],\
                         STR);\
          exit(EXIT_FAILURE);\
        }\
        VAR = (unsigned int) readint

static Matrixtype addMatrixtype(int line,Matrixtype a,Matrixtype b)
{
  if (a > (~0ULL) - b) {
    fprintf(stderr,"line %d: ",line);
    fprintf(stderr,MATRIXFORMAT,a);
    fprintf(stderr," ");
    fprintf(stderr,MATRIXFORMAT,b);
    fprintf(stderr," leads to overflow\n");
    exit(EXIT_FAILURE);
  }
  return a+b;
}

Matrixtype alignsrecursive(Matrixtype i,Matrixtype j)
{
  Matrixtype addtmp;

  countcalls++;
  if (i == 0 || j == 0)
    return 1;
  addtmp = addMatrixtype(__LINE__,alignsrecursive(i,j-1),
                                  alignsrecursive(i-1,j));
  addtmp = addMatrixtype(__LINE__,addtmp,alignsrecursive(i-1,j-1));
  return addtmp;
```

```c
}

Matrixtype alignstabulated(Matrixtype m, Matrixtype n)
{
  Matrixtype tmpval, we, nw, i, j, *alignscolumn;

  alignscolumn = (Matrixtype *) malloc(sizeof(Matrixtype) * (m+1));
  if (alignscolumn == NULL) {
    fprintf(stderr,"line %d: malloc(%d) failed\n",
                    __LINE__,(int) (sizeof(Matrixtype) * (m+1)));
    exit(EXIT_FAILURE);
  }
  for (i=0;i<=m;i++)
    alignscolumn[i] = 1;
  for (j=0;j<n;j++) {
    for (nw = 1, i=1; i<=m; i++) {
      countaccesses += 3;
      we = alignscolumn[i];
      tmpval = addMatrixtype(__LINE__,we,nw);
      alignscolumn[i] = addMatrixtype(__LINE__,tmpval,alignscolumn[i-1]);
      nw = we;
    }
  }
  return alignscolumn[m];
}

int main(int argc, char *argv[])
{
  int readint;
  unsigned int m, n;
  char mode;

  if (argc != 4) {
    fprintf(stderr,"Usage: %s mode m n\n"
                   "mode can be r (recursive) or t (tabulated)\n", argv[0]);
    return EXIT_FAILURE;
  }

  /* scan mode */
  if (sscanf(argv[1],"%c",&mode) != 1) {
    fprintf(stderr,"Usage: %s mode m n\n"
                   "first argument must be character r (recursive) or "
                   "t (tabulated)\n", argv[0]);
    exit(EXIT_FAILURE);
  }
  if (!(mode == 'r' || mode == 't')) {
    fprintf(stderr, "mode %c does not equal r or t\n", mode);
    exit(EXIT_FAILURE);
  }

  READINT(2,m,"second");
  READINT(3,n,"third");

  if (mode == 'r') {
    printf("%u %u ", m, n);
    printf(MATRIXFORMAT, alignsrecursive(m, n));
    printf(" (%llu calls)\n", countcalls);
  }
  else { /* mode == 't' */
    printf("%u %u ", m, n);
    printf(MATRIXFORMAT, alignstabulated((Matrixtype) m, (Matrixtype) n));
    printf(" (");
```

```
    printf(MATRIXFORMAT, countaccesses);
    printf(" accesses)\n");
  }
  return EXIT_SUCCESS;
}
```

Ruby-Variante

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

require 'optparse'

# class to calculate Aligns(i,j) recursively
class Aligns_recursive
  attr_accessor :m
  attr_accessor :n
  attr_accessor :count

  def initialize(m, n)
    @m = m
    @n = n
    @count = 0
  end

  def recursion(m,n)
    @count += 1
    if (m > 0) and (n > 0)
      return recursion(m-1, n) + recursion(m, n-1) + recursion(m-1, n-1)
    else
      return 1
    end
  end

  def calculate
    return recursion(@m, @n)
  end

  def type
    return "calls"
  end
end

# class to calculate Aligns(i,j) using a matrix
class Aligns_iterative
  attr_accessor :m
  attr_accessor :n
  attr_accessor :count

  def initialize(m, n)
    @m = m
    @n = n
    @count = 0
  end

  def calculate
    aligns_column = Array.new

    for i in 0..@m
      aligns_column[i] = 1
    end
    for j in 0..(@n - 1)
      nw = 1
```

```
      for i in 1..@m
        @count += 3
        we = aligns_column[i]
        aligns_column[i] = we + nw + aligns_column[i-1]
        nw = we
      end
    end
    return aligns_column[m]
  end

  def type
    return "accesses"
  end
end

align_rec = false

opts = OptionParser.new() do |opts|
  opts.banner = "Usage: #{$0} [options] m n\n"+
    "With m and n are positive numbers below 15."

  opts.separator ""
  opts.separator "options:"
  opts.on("-r","--recursive","Run recursive algorithm") do |v|
    align_rec = v
  end
end

opts.parse!

# main() analogon
if ARGV.length != 2
  STDERR.puts opts.help
  exit 1
else
  begin
    m=Integer(ARGV[0])
    n=Integer(ARGV[1])
  rescue => e
    STDERR.puts e
    STDERR.puts opts.help
  end
  if not m.between?(0,15) or not n.between?(0,15)
    STDERR.puts "m or n out of bounds"
    STDERR.puts opts.help
  end
end

# get class to use
if (align_rec)
  aligns_class = Aligns_recursive.new(m, n)
else
  aligns_class = Aligns_iterative.new(m, n)
end

# calculate Aligns(m,n)
result = aligns_class.calculate

# print result
puts "#{m} #{n} #{result} (#{aligns_class.count} #{aligns_class.type})"
```

Lösung zu Aufgabe 2.2:

1. a) Die Menge aller graden natürlichen Zahlen, die kleiner als $2k$ sind.

   b) die Menge aller Paare (i, j) für die gilt:

   - i ist kleiner als j,
   - i und j liegen beide im Wertebereich zwischen 1 und n,
   - die Basen aus w an den Positionen i, j sind gepaart
   - die Anzahl der Basen zwischen i und j ist mindestens l.

2. a) $H(x, y) := \{i \mid 1 \leq i \leq |w|, w[i] \neq v[i]\}$

   b) $M_k(w, v) := \{(i, j) \mid 1 \leq i \leq |w| - k + 1, 1 \leq j \leq |v| - k + 1, w[i \ldots i + k - 1] = v[j \ldots j + k - 1]\}$

Lösung zu Aufgabe 2.3:

Ruby Solution

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

Matchpos = Struct.new(:l, :r)

def lcs_sequence(u,v)
  return lcs_sequence_rec(u,0,v,0)
end

def lcs_sequence_rec(u,i,v,j)
  if i == u.length or j == v.length
    return []
  end
  if u[i] == v[j]
    return [Matchpos.new(i,j)] + lcs_sequence_rec(u,i+1,v,j+1)
  else
    s1 = lcs_sequence_rec(u,i+1,v,j)
    s2 = lcs_sequence_rec(u,i,v,j+1)
    if s1.length > s2.length
      return s1
    else
      return s2
    end
  end
end

if ARGV.length != 2
  STDERR.puts "Usage: #{$0} <string1> <string2>"
  exit 1
end
s1 = ARGV[0]
s2 = ARGV[1]
puts "lcs(#{s1},#{s2})="
lcs = lcs_sequence(s1,s2)
puts lcs.map {|m| "(#{m.l+1},#{m.r+1})"}.join(",")
puts lcs.map {|m| s1[m.l].to_s}.join("")
```

C Solution

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

```c
#include <stdlib.h>

unsigned long lcs_length(const char *u, const char *v)
{
  if (*u == '\0' || *v == '\0')
  {
    return 0;
  } else
  {
    if (*u == *v)
    {
      return 1 + lcs_length(u+1, v+1);
    } else
    {
      unsigned long lcs1 = lcs_length(u+1,v);
      unsigned long lcs2 = lcs_length(u,v+1);
      return lcs1 < lcs2 ? lcs2 : lcs1;
    }
  }
}

int main(int argc, char *argv[])
{
  if (argc != 3)
  {
    fprintf(stderr,"Usage: %s <sequence1> <sequence2>\n",argv[0]);
    return EXIT_FAILURE;
  }
  printf("lcs(%s,%s)=%lu\n",
          argv[1],argv[2],lcs_length(argv[1],argv[2]));
  return EXIT_SUCCESS;
}
```