S. Kurtz, T. Schwabe,
G. Gonnella, D. Willrodt

Universität Hamburg
Zentrum für Bioinformatik

Programmierung für Naturwissenschaften
Sommersemester 2015
Übungen zur Vorlesung: Ausgabe am 23.04.2015

Punkteverteilung: Aufgabe 4.1: 4 Punkte, Aufgabe 4.2: 6 Punkte

Abgabe bis zum 29.04.2015, 10:00 Uhr.

Aufgabe 4.1 Implementieren Sie ein C-Programm `revcomp.x`, welches eine Datei mit einer DNA-Sequenz einliest und zu dieser das reverse Komplement in eine neue Datei schreibt. Die Datei enhält nur die Zeichen `A`, `C`, `G` und `T`. Das reverse Komplement einer DNA-Sequenz ist die Sequenz des komplementären DNA-Stranges in der Doppenhelixstruktur. Jede Base wird mit ihrer Watson-Crick-Komplement ersetzt (`A`→`T`, `C`→`G`, `G`→`C`, `T`→`A`) und die Sequenz wird von hinten nach vorne gelesen.

Dabei soll die Formatierung, insbesondere die Zeilenbreite, erhalten bleiben. Der Dateiname der neuen Datei soll dem der Eingabedatei entsprechen, erweitert um den Suffix `.rc`. Um die Sequenz in umgekehrter Reihenfolge zu bearbeiten, müssen Sie diese vollständig einlesen und speichern. Dazu muss zunächst die Größe der Eingabedatei ermittelt werden. Verwenden Sie dazu die folgende Funktion aus der Vorlesung:

```
unsigned long file_size(const char *inputfile)
{
  long filesize;
  FILE *fp = fopen(inputfile,"rb");

  assert(fp != NULL);
  fseek (fp, 0, SEEK_END); /* jump to end of file */
  filesize = ftell(fp); /* read current position */
  fclose(fp);
  assert(filesize >= 0);
  return (unsigned long) filesize;
}
```

Eine Beispiel-Datei mit einer Sequenz zum Testen finden Sie in STiNE (`ecoli.seq`). Die erwartete Ausgabe finden Sie in der Datei `expected_output`. Der Aufruf von `make test` soll fehlerfrei beenden.

Aufgabe 4.2 Wir wollen am Anfang die Aufgabenstellung ausführlich motivieren. Biologische Sequenzen, wie z.B. DNA- oder Protein-Sequenzen, werden oft in Textdateien im Fasta-Format gespeichert. Diese bestehen aus Kopf-Zeilen, die mit dem Zeichen > beginnen und Sequenzzeilen, die nicht mit dem Zeichen > beginnen. Die Kopfzeilen beschreiben die Sequenz (z.B. durch Identifikatoren in Datenbanken), die nach einem Zeilenumbruch direkt auf die Kopfzeile folgt und meist aus mehreren Zeilen besteht. Hier ein Beispiel für den Inhalt einer Datei, die aus drei DNA-Sequenzen der Längen 28, 30 und 17 besteht:

```
>gi|5587835|Arabidopsis thaliana cDNA clone IFA2
GCGGCGGTGGCGGCGA
```

```
TGTGATGATTAT
>gi|4714049|Arabidopsis thaliana cDNA clone 88
ATGCACACTCAAAAATGG
TCCCTACTTTGC
>gi|4714048|Arabidopsis thaliana cDNA clone 87-1
GTGACTGTTGATCTCAA
```

Für die Analyse der Sequenzen in einer Fasta-Datei (wie die Suche nach Mustern oder der Sequenzvergleich) sind nur die Sequenzteile wichtig, während die Kopfzeilen meist bei der Ausgabe eine Rolle spielen. Daher trennt man den Inhalt einer Fasta-Datei in den Beschreibungsteil, der nur aus den Kopfzeilen besteht und den Sequenzteil, der nur aus den Sequenzen besteht. Technisch bildet man aus beiden Teilen jeweils einen String: Der Beschreibungsstring enthält alle Kopfzeilen, getrennt durch ein Newline-Zeichen, also in unserem Beispiel

```
>gi|5587835|Arabidopsis thaliana cDNA clone IFA2
>gi|4714049|Arabidopsis thaliana cDNA clone 88
>gi|4714048|Arabidopsis thaliana cDNA clone 87-1
```

Die Sequenzen werden (ohne Leerzeichen) jeweils mit einem Separatorzeichen, z.B. `$`, zwischen zwei aufeinanderfolgen Sequenzen konkateniert. Man erhält eine Sequenz T, die in unserem Beispiel wie folgt aussieht:

```
GCGGCGGTGGCGGCGATGTGATGATTAT$ATGCACACTCAAAAATGGTCCCTACTTTGC$GTGACTGTTGATCTCAA
```

Wenn man die Positionen in T beginnend mit 0 indiziert, dann kommt das `$`-Zeichen an der Position 28 und an der Position $28 + 1 + 30 = 59$ vor. Die konkatenierte Sequenz hat insgesamt die Länge $28 + 1 + 30 + 1 + 17 = 77$.

Da die DNA-Sequenzen, z.B. ganze Genome, viele Milliarden Zeichen enthalten, ist es sinnvoll, sie komprimiert darzustellen. Da eine DNA-Sequenz aus den vier Basen `A`, `C`, `G`, `T` und dem Separatorzeichen besteht, wäre eine Repräsentation der Sequenz durch eine Folge von ganzen Zahlen möglich, die jeweils durch $\lceil \log_2 5 \rceil = 3$ Bits dargestellt werden. Allerdings kommt das `$`-Zeichen im Vergleich zu den vier Basen sehr selten vor und das dafür notwendige zusätzliche Bit möchte man vermeiden. Daher benutzt man auch für das `$`-Zeichen eine ganze Zahl $\lambda \in \{0, \ldots, 3\}$ (die sich durch 2 Bits darstellen läßt). Damit man die `$`-Zeichen in der 2-Bit-Repräsentation der Sequenz erkennen kann, speichert man zusätzlich die Positionen, an denen das `$`-Zeichen vorkommt, in einer Menge S ab. Die Elemente von S nennen wir *Separatorpositionen*.

In unserem Fall besteht S aus den beiden ganzen Zahlen 28 und 59. Allgemein gilt: wenn man $k > 0$ Sequenzen der Längen $n_0, n_1, \ldots, n_{k-1}$ hat, dann erhält man ein Menge $S = \{p_0, \ldots, p_{k-2}\}$ von $k-1$ nicht-negativen ganzen Zahlen $p_j = j + \sum_{i=0}^{j} n_i$ für alle j, $0 \leq j \leq k-2$. Offensichtlich gilt $p_{j-1} < p_j$ für alle j, $1 \leq j \leq k-2$.

Aufgabenstellung: Sie sollen in einer Datei `intset-simple.c` einen Datentyp `IntSet` implementieren, der eine aufsteigend sortierte Menge S von nicht-negativen ganzen Zahlen repräsentiert und mit den folgenden Funktionen zur Verfügung stellt:

```
/* The <IntSet> class, used to store sorted integer sets. */
typedef struct IntSet IntSet;

/* Return a new <IntSet> Object with space allocated for <nofelements> elements
   [0,<maxvalue>] */
IntSet *intset_new(unsigned long maxvalue, unsigned long nofelements);

/* Returns the size needed to store <nofelements> integers smaller equal
   <maxvalue> */
size_t intset_size(unsigned long maxvalue, unsigned long nofelements);

/* Free the memory of <intset> */
void intset_delete(IntSet *intset);

/* Add <elem> to <intset>. Fails if <elem> is larger <maxvalue> or <intset>
   already contains <nofelements> elements. */
void intset_add(IntSet *intset, unsigned long elem);

/* True if <elem> is stored in <intset> */
bool intset_is_member(const IntSet *intset, unsigned long elem);

/* Return the index/number of the smallest element that is larger then
   <value> from <intset> or <nofelements> if there is no such element. */
/* Returns 0 for this exercise */
unsigned long intset_number_next_larger(const IntSet *intset,
```

Hinweise:

- Offensichtlich ist `IntSet` ein Typsynonym für eine Struktur `struct IntSet`. Diese soll u.a. ein dynamisch allokiertes Array `elements` über dem Basistyp `unsigned long` enthalten, in dem die Elemente der Menge S gespeichert werden können. Die Struktur enthält natürlich noch weitere Werte.
- Sie brauchen sich nicht darum zu kümmern, woher die Elemente von S kommen, sondern lediglich die hier genannten Funktionen implementieren.
- `intset_new` ist der Konstruktor für den Datentyp `IntSet`, der zunächst noch keine Elemente enthält. Dabei gibt `maxvalue` eine obere Schranke für die Werte an, die gespeichert werden sollen. `nofelements` gibt die Anzahl der Elemente an, die gespeichert werden sollen. Der Parameter `maxvalue` wird erst für eine effiziente Implementierung von `IntSet` (nächstes Übungsblatt) benötigt, ist aber aus Gründen der Kompatibilität hier schon aufgeführt. Da ein Set von Zahlen keine Duplikate enthalten kann, muss `nofelements` $<=$ `maxvalue` sein.
- `intset_size` gibt die Grösse der Repräsentation in Bytes zurück. Durch die Verwendung der gleichen Parameter wie beim Konstruktor kann die Grösse ermittelt werden, ohne die Repräsentation selbst zu erzeugen.
- `intset_delete` ist der Destruktor für den Datentyp `IntSet` und gibt den Speicherplatz für einen `IntSet`-Wert wieder frei.
- `intset_add` fügt ein neues Element zu der Menge hinzu. Dabei muss das neue Element echt grösser sein als alle bisher eingefügten Elemente und kleiner gleich dem `maxvalue`-Wert, der dem Konstruktor übergeben wurde.
- `intset_is_member` liefert genau dann `true` zurück, wenn der übergebene Wert in der re-

präsentierten Menge enthalten ist. Die Laufzeit dieser Funktion soll in $O(\log_2(k-1))$ liegen, wenn die Menge $k-1$ Elemente enthält.

- `intset_number_next_larger` wird in der Aufgabe noch nicht benötigt. Da das Hauptprogramm jedoch diese Funktion aufruft, implementieren Sie `intset_number_next_larger` als konstante Funktion, die immer `0` zurückliefert.
- Verwenden Sie das `assert`-Makro, um Bedingungen, die an der entsprechenden Stelle des Programms erfüllt sein müssen, zu überprüfen.
- Die Materialien zu dieser Übung enthalten eine Datei `intset.h`, ein Hauptprogramm `intset-main.c`, ein Shell-Skript `intset-run.sh` und ein `Makefile`. Durch `make` wird der Quelltext compiliert und eine ausführbare Datei `intset.x` erzeugt. `make test` ruft `intset-run.sh` auf. Wenn alles gut geht, benötigen die 57 Testfälle etwa 10 Sekunden.

Anwendung von `IntSet`: Der Datentyp `IntSet` ist ein wichtiger Teil einer 2-Bit-Repräsentation einer Menge von DNA-Sequenzen. Wenn man wissen möchte, ob an einer Position i in der repräsentierten DNA-Sequenz ein Separatorzeichen steht, extrahiert man aus der Repräsentation T das i-te Paar von Bits, nennen wir es b. Dieses stellt eine Zahl zwischen 0 und 3 dar. Ist $b \neq \lambda$, dann repräsentiert b eine vor drei Basen. Ist $b = \lambda$, dann repräsentiert b ein Separatorzeichen genau dann, wenn $i \in S$ gilt. Wenn b kein Seperatorzeichen repräsentiert, dann steht es für eine Base. Da S durch den Datentyp `IntSet` repräsentiert wird, gilt $i \in S$ genau dann, wenn die Funktion `intset_is_member` für die Position i den Wert `true` zurückliefert. Falls $i \notin S$, stellt b eine der vier Basen dar.

Zusatzfragen:

- Nehmen Sie an, Sie kennen die Häufigkeitsverteilung der Zeichen in der zu repräsentierenden Sequenz T. Welcher Wert sollte dann sinnvollerweise für λ gewählt werden.
- Nehmen Sie an, alle zu repräsentierenden Sequenzen haben die gleiche Länge. Wie kann man dann die Funktion `intset_is_member` implementieren?

Die Lösungen zu diesen Aufgaben werden am 30.05.2015 besprochen.