Übungen zur Vorlesung
Algorithmen auf Sequenzen

TU Dortmund, SS 2008
Prof. Dr. Sven Rahmann

Blatt 2 und 3 vom 14.04.2008
Abgabe am Fr 25.04.2008 in der Vorlesung

Achtung! Die Veranstaltungen am 18.04. und 21.04. müssen leider ausfallen.
Vergnügen Sie sich in der Zwischenzeit mit diesen Aufgaben.
Die nächste Vorlesung findet am 25.04.2008 statt.

Aufgabe 1 Beschreibe einen möglichst effizienten Algorithmus, um aus der Präfix-Funktion des KMP-Algorithmus die Übergangsfunktion des DFA zu berechnen. Gib die Laufzeit abhängig von Musterlänge m und Alphabetgröße $\sigma = |A|$ an.

Aufgabe 2 Gegeben sind zwei Strings s, t gleicher Länge n. Gib einen Algorithmus an, mit dem man in $O(n)$ Zeit entscheiden kann, ob t eine zyklische Permutation von s ist, und der ggf. zyklische die Verschiebung von t gegenüber s angibt.

Aufgabe 3 Erläutere, wie sich **shift-and** und **shift-or** auf verallgemeinerte Strings (siehe Blatt 1) anwenden lassen. Gib explizit die Maskenerstellung und den Matching-Algorithmus von **shift-or** mit dieser Verallgemeinerung an. Welche Invariante ist nach jedem Schritt erfüllt?

Aufgabe 4 Implementiere möglichst viele der besprochenen Algorithmen.

- naiv vorwärts (präfixbasiert), naiv rückwärts (suffixbasiert)
- DFA-basiertes matching (die δ-Funktion des Automaten kann naiv oder effizienter gemäß Aufgabe 1 berechnet werden)
- KMP
- shift-and, shift-or
- BM-Horspool, BM-Sunday

Sowohl Texte als auch Muster sollen als `byte[]` dargestellt werden. Verwende durchgängig folgende Funktions-Signatur (hier am Beispiel von KMP in Java):

```
int KnuthMorrisPratt(final byte[] T, final byte[] P,
                     final boolean decisionOnly, final ArrayList<Integer> matches)
```

Der Rückgabewert ist die Anzahl der Vorkommen von `P` in `T`. Wenn `decisionOnly` gesetzt ist, bricht der Algorithmus nach dem ersten Vorkommen ab und gibt 1 zurück, falls das Muster mindestens einmal im Text vorkommt. Wenn `matches` nicht `null` ist, werden darin die Positionen der Vorkommen in aufsteigender Reihenfolge gespeichert.

Aufgabe 5 Implementiere eine Funktion, die zufällige Texte und Muster erzeugt. Die Funktion bekommt die Länge des zu erzeugenden Textes n und die Alphabetgröße σ übergeben und nimmt an, dass alle Zeichen in $\{0, \ldots, \sigma - 1\}$ die gleiche Wahrscheinlchkeit haben.

```
byte[] createRandomText(final int n, final int sigma, byte[] T)
```

Wenn das übergebene Array `T` groß genug ist, wird es überschrieben und zurückgegeben. Eventuell zusätzlche Zeichen in `T` werden mit dem ungültigen Wert -1 gefüllt. Ist `T` zu klein oder `null`, wird ein neues Array der Größe n angelegt.

(*) Optional: Alternativ zu σ kann ein Wahrscheinlichkeitsvektor `double[] p` mit `p.length` $= \sigma$ übergeben werden, der für jedes Zeichen in $\{0, \ldots, \sigma - 1\}$ eine Wahrscheinlichkeit spezifiziert. Am effizientesten wird in diesem Fall ein zufälliger Text nach der sogenannten Alias-Methode erzeugt.

Aufgabe 6 Wir wollen möglichst viele der besprochenen Algorithmen in der Praxis miteinander vergleichen. Zum Testen der einzelnen Algorithmen legen wir folgende Parameter zugrunde.

- Textlänge $n = 100$ MB; wenn der verwendete Rechner zu wenig Speicher hat, reduziere ggf. n, aber versuche, möglichst lange Texte zu verarbeiten. Ist genug RAM vorhanden, versuche auch $n = 1$ GB. Gemessene Laufzeiten sollten immer auf 1 MB normalisiert werden (also hier durch 100 geteilt werden).
- Patternlänge $m \in \{2, \ldots, 256\}$; für $m \geq 32$ muss nicht mehr jede einzelne Länge genommen werden. Gemessene Laufzeiten sollten durch m geteilt werden (Laufzeit pro MB Text, pro Pattern-Zeichen).
- Alphabetgröße $\sigma \in \{2, \ldots, 128\}$; damit passt jedes Zeichen des Alphabets $\{0, \ldots, \sigma - 1\}$ immer in die untersten 7 bits eines Java `byte`.

Miss nun die Laufzeiten der implementierten Algorithmen; dies kann CPU-Zeit oder Realzeit sein. Die Vorverarbeitung des Musters gehört mit zur Laufzeit. Vorschlag zum systematischen Vorgehen:

```
for each pattern length m do
  for each alphabet size sigma do
    for sample = 1 .. 100 (or 1000)
      randomly create T of length n, P of length m
      for attempt = 1 .. 20
        for each algorithm ALG
          t[attempt]{ALG} := normalized matching time for ALG in this attempt
      t[sample]{ALG} := min t[attempt]{ALG} (minimum over all attempts)
    tAvg[m][sigma]{ALG} := average of t[sample]{ALG} over all samples
    A[m][sigma] := algorithm with mininal tAvg[m][sigma]{ALG} over all algorithms
draw a color-coded map of A[m][sigma] (axes: m and sigma)
```

Die Farbcodierung der Karte des besten Algorithmus für jede Parameterkombination sollte anhand der Farbe zeigen, welcher Algorithmus “gewonnen” hat. Denke dir andere auch Visualisierungsmöglichkeiten der gewonnenen Daten aus.