

# Exakte Suche/ Pattern Matching

K. Dührkop, M. Fleischauer

Lehrstuhl für Bioinformatik  
Fakultät für Mathematik und Informatik  
Friedrich-Schiller-Universität Jena

# Exakte Suche

- Suche nach einem String in einem anderen.
- Definition:  
Gegeben sei eine Sequenz  $a = a_1 \dots a_n$  und ein Muster  $b = b_1 \dots b_m$ . Ein Vorkommen von  $b$  in  $a$  (an Position  $i$ ) ist eine Subsequenz  $a_i \dots a_{i+m-1}$  von  $a$ , die exakt gleich  $b$  ist. Exakte Suche ist das Problem die/ein Vorkommen von  $b$  in  $a$  zu finden.

# Anwendungen in der Bioinformatik

- direkt: Suche nach exakten Teilsequenzen
- deutlich schneller als nicht-exakte Suche, deshalb für Approximation nicht-exakter Suche.
- Sequenzen können Nukleotid- oder AS-Sequenzen sein, aber durchaus auch “lesbare” Texte

## Das Naive Verfahren

- Prinzip: das Muster wird an allen Positionen der Sequenz von Neuem mit der Sequenz verglichen.

```

          1           2           3
123456789012345678901234567890123
anna mag banane lieber als ananas
ananas
ananas
  ananas ...
      ananas
      ananas ...
          ananas
          ananas ...
```

- Komplexität: Worst-Case  $O(n \cdot m)$ . Wird das Muster jeweils nur soweit wie nötig verglichen, ergibt sich in der Praxis oft  $O(n + m)$  im Mittel.

# Knuth-Morris-Pratt Algorithmus

- Idee
  - Vermeide Teile der Sequenz mehrfach zu lesen.
  - Verschiebe das Muster soweit wie möglich, statt wie im naiven Verfahren immer nur um eine Position.
- Beispiel

```
          1           2           3
123456789012345678901234567890123
anna mag banane lieber als ananas
ananas
  ananas ...
      ananas
        ananas...
                                ananas ...
```

- Verschiebung lässt sich aus Muster berechnen, dann  $O(n)$

# Knuth-Morris-Pratt Algorithmus

- Berechnung der nächsten Position im Muster
  - $next[j] = 1 + \text{Länge des längsten echten Suffixes von } b_1 \dots b_{j-1}, \text{ das zugleich Präfix von } b \text{ ist}$
  - $next[1] = 0$  (Stopper-Wert)
- Konstruktion des next-Arrays durch „Suche des Musters in sich selbst.“
- Verfahren zur Suche von  $b$  in  $a$  (Längen  $m$  und  $n$ )

```
int i=1, j=1; // Zeiger in Sequenz(i) und Muster(j)
while (i<=n && j<=m)
    if (j==0 || a[i] == b[j])
        i++; j++;
    else
        j = next[j];
if (i<=n) gefunden an Position i-m else nicht gefunden
```

# Boyer-Moore Algorithmus

- Idee
  - viele Textzeichen kommen im Muster nicht oder nur selten vor
  - das lässt sich ausnutzen, um nicht mehr alle Textzeichen zu betrachten
- Prinzip
  - das Muster wird von rechts beginnend mit der Sequenz verglichen, aber wie bisher von links nach rechts über die Sequenz geschoben
  - falls das Zeichen der Sequenz gar nicht im Muster auftaucht, kann das Muster komplett um eine Musterlänge verschoben werden.(Bad-Character)
  - sonst lässt sich immer noch berechnen wie weit das Muster bei einem Mismatch verschoben werden kann.(Good-Suffix)

# Bad-Character-Heuristik

Hoola-Hoola girls like Hooligans

Hooligan          

    Hooligan      

        Hooligan   

            Hooligan 

                Hooligan



# (Strong) Good-Suffix-Heuristik

```
reinesupersauersupesupersupe  
supersupe  
    supersupe  
        supersupe  
            supersupe
```

# Boyer-Moore Algorithmus

- $\mathit{delta}_{BCH}$ -Tabelle

$$\mathit{delta}_{BCH}(c) = \begin{cases} m - j & \text{falls } b_j \text{ rechtestes Vorkommen von } c \text{ in } b \text{ ist} \\ m & \text{falls } c \text{ nicht in } b \end{cases}$$

- $\mathit{delta}_{GSH}$ -Tabelle

$$\mathit{delta}_{GSH}(j) = \begin{cases} j - k & \text{falls } b_{k+1} \dots b_{k+m-j} \text{ mit } b_k \neq \\ & b_j \text{ rechtestes Vorkommen von } b_{j+1} \dots b_m \text{ ist} \\ m & \text{falls } b_{k+1} \dots b_{k+m-j} \text{ mit } b_k \neq b_j \text{ nicht in } b \text{ vorkommt} \end{cases}$$

# Boyer-Moore Algorithmus

- Algorithmus zur Suche von  $b$  in  $a$  (Längen  $m$  und  $n$ )

```
int i=m, j=m; // Zeiger in Sequenz und Muster
while (j>=1 && i<=n)
    if (a[i] == b[j])
        i--; j--;
    else
        i+=max(m-j+1, deltaBCH(a[i]), deltaGSH(j))
        j=m
return i+1;
```

# Aufgaben

- Versionskontrolle mit Git
- Projektmanagement mit Gradle
- Implementierung der Klassen mit den entsprechenden Algorithmen (Naiv, KMP, BM)
- API Dokumentation mit javadoc
- Kommandozeilenprogramm (CLI) + README/Hilfe
- Testläufe mit Protokollierung der Ergebnisse (Testdaten und Aufgabenblatt auf der Kurswebsite)

# Anforderungen an die Implementierung

- Package für Sequenzen
  - Gemeinsame API für Sequenzen
  - Klassen für ASCII-, DNA- und Aminosäuresequenzen
  - besitzen Name und die eigentliche Sequenz
  - zusätzlich für DNA-Sequenzen
    - Rückgabe der komplementären Sequenz
    - Translatieren in Aminosäuresequenz (3 Leseraster!)
- Package für I/O
  - lesen und schreiben im Fasta-Format

# Anforderungen an die Implementierung

- Package für die Textsuche
  - Gemeinsame API fuer Textsuchealgorithmen
  - Klassen für Naive Textsuche, KMP, Boyer-Moore
  - Nehmen Text und Muster entgegen
  - erlauben Suche des Musters in dem Text
    - jeweils das nächste Auftreten (Position) des Musters im Text (bis das Ende des Textes erreicht ist)
    - Anzahl der Vorkommen des Musters im Text
- Package für main() / CLI
  - Parameter für Text- und Muster-Datei
  - Parameter für den Suchalgorithmus
  - Parameter für Typ der Sequenzen