Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Moritz Kobitzsch, Dennis Schieferdecker

7. Übungsblatt zu Algorithmen II im WS 2011/2012

http://algo2.iti.kit.edu/AlgorithmenII.php
{kobitzsch,sanders,schieferdecker}@kit.edu

Aufgabe 1 (*Rechnen: Kompression*)

a) Führen Sie eine Kompression nach *Lempel-Ziv* auf der Zeichenkette $s =$ `abababcabcc` aus.

b) Führen Sie eine Dekompression auf der codierten Zeichenkette $c =$ `1,2,3,5,4,2` aus. Sie wissen, dass die ursprüngliche Zeichenkette über dem Alphabet $\{$`a`, `b`$\}$ aufgebaut ist.

c) Durch einen Fehler in der Datenübertragung wurde nur jedes zweite Zeichen einer mit *Lempel-Ziv* komprimierten Zeichenkette übertragen. Glücklicherweise wurde auch das verwendete Wörterbuch mitübertragen. Leider fehlt auch hier jeder zweite Eintrag. Rekonstruieren Sie die codierte Zeichenkette und das Wörterbuch und geben Sie die unkomprimierte Zeichenkette an.

Die übermittelte Zeichenkette lautet $c = 1, ?, 2, ?, 5$. Das übermittelte Wörterbuch enthält die Einträge $D = \{(1, ?), (2, \texttt{b}), (3, ?), (4, \texttt{aab}), (5, ?), (6, \texttt{aabb})\}$. Sie wissen außerdem, dass die ursprüngliche Zeichenkette über dem Alphabet $\{$`a`, `b`$\}$ aufgebaut wurde.

Aufgabe 2 (*Rechnen+Analyse: Suche in Strings*)

a) Zur Ausführung des KMP-Algorithmus muss zunächst ein sogenanntes *border-array* berechnet werden. Geben Sie das *border-array* für das Suchmuster $p =$ `abacababc` an.

b) Führen Sie den KMP-Algorithmus auf dem Text $t =$ `abacababbabacababc` mit obigem Suchmuster durch.

c) Wie oft muss der KMP-Algorithmus ein Muster p der Länge $|p|$ maximal an einen Text t der Länge $|t|$ anlegen, falls p nicht in t vorkommt? Wie oft minimal? Geben Sie das Ergebnis in Abhängigkeit von $|p|$ und $|t|$ an. Geben Sie außerdem jeweils ein Beispiel für p und t an.

d) Zeigen oder widerlegen Sie:
Das *border-array* kann keine drei aufeinanderfolgenden Einträge enthalten, die jeweils um eins kleiner sind als ihr Vorgänger (Beispiel: $\text{border}[10] = 3,\ \text{border}[11] = 2,\ \text{border}[12] = 1$).

Aufgabe 3 (*Rechnen: Burrows-Wheeler-Transformation*)

a) (*) Bestimmen Sie die Entropie von Zeichenkette $s =$ `abababababab`.

b) Führen Sie die *Burrows-Wheeler-Transformation* auf Zeichenkette s aus. Wie groß ist jetzt die Entropie der Zeichenkette?

c) Führen Sie eine *Move-to-Front* Kodierung auf dem Ergebnis durch.
(das Abschlusszeichen `$` aus der BWT muss bei der Kompression nicht berücksichtigt werden)

d) Vergleichen Sie das Ergebnis mit einer direkten *Move-to-Front* Kodierung von s. Wie groß ist jeweils die Entropie der beiden kodierten Zeichenketten?

e) Führen Sie eine inverse *Burrows-Wheeler-Transformation* auf Zeichenkette $s^{BWT} =$ `bc$aab` aus.

Aufgabe 4 (*Rechnen: Suffixarrays und DC3-Algorithmus*)

Gegeben sei die Zeichenkette $s =$ `aberakadabera`.

a) Geben Sie den Suffixbaum für s an.

b) Geben Sie das Suffixarray für s an.

In der Vorlesung haben Sie einen Linearzeitalgorithmus zur Konstruktion von Suffixarrays kennengelernt. Dieser ist unter dem Namen *DC3-Algorithmus* bekannt. Im Folgenden soll der Algorithmus Schritt für Schritt per Hand ausgeführt werden.
Die Suffixe von s werden zunächst in drei Sequenzen $C^k = \langle s_i \mid (i \bmod 3) = k \rangle$ für $k \in \{0, 1, 2\}$ aufgeteilt. Danach müssen die Sequenzen C^0 und $C^{12} = C^1 \cup C^2$ lexikographisch sortiert werden.

Sortierung von C^{12}:

c) Geben Sie die Tripelsequenzen $R^k = \langle s[i..i+2] \mid (i \bmod 3) = k \rangle$ für $k \in \{1, 2\}$ an. Für $i \geq |s|$ gelte $s[i] =$ \$ (Auffüllen mit zusätzlichen Abschlusszeichen).

d) Bestimmen Sie den *Rang* der Tripel von $R^{12} = R^1 \circ R^2$. Sortieren Sie dazu die Tripel und entfernen mehrfache Vorkommnisse. Die Position eines Tripels in dieser Sortierung gibt seinen Rang an.

e) Die berechneten Ränge definieren eine eindeutige Bezeichnung für jedes Tripel in R^{12}. Drücken Sie R^{12} mit Hilfe dieser Ränge aus. Diese Darstellung ergibt die Zeichenkette s^{12}. Muss der DC3-Algorithmus eine Rekursion ausführen?

f) Geben Sie das Suffixarray SA^{12} für s^{12} von Hand an (unabhängig, ob der DC3-Algorithmus eine Rekursion durchführt). Vergewissern Sie sich, dass SA^{12} eine Sortierung von C^{12} beschreibt.

Sortierung von C^0:

g) Erstellen Sie eine Zuordnung $rank$, die jedem i mit $s_i \in C^{12}$ den Index von s_i in der sortierten Sequenz C^{12} zuweist. Für alle anderen i sei $rank(i) = 0$.

   Formale Berechnung von $rank$ mit Hilfe des Suffixarray SA^{12} nach:

$$\begin{array}{lcll} rank[3 \cdot (\mathrm{SA}^{12}[i]) \quad\quad + 1] & = & i & \mathrm{SA}^{12}[i] < |C^1| \\ rank[3 \cdot (\mathrm{SA}^{12}[i] - |C^1|) + 2] & = & i & \mathrm{SA}^{12}[i] \geq |C^1| \end{array}$$

   Alle anderen Werte von $rank[i]$ können gleich 0 gesetzt werden.

h) Erstellen Sie Tupel $(s[i], rank[i{+}1])$ f.a. $s_i \in C^0$ und sortieren diese lexikographisch. Vergewissern Sie sich, dass diese Sortierung einer Sortierung von C^0 entspricht.

Nachdem C^0 und C^{12} sortiert worden sind, kann das Suffixarray von s bestimmt werden:

i) Führen Sie eine Mischen-Operation auf C^0 und C^{12} aus. Die resultierende Sequenz wird mit C bezeichnet. Es gelten folgende Sortierkriterien:

$$s_i \leq s_j \iff \begin{cases} (s[i], \quad\quad rank[i+1]) & \leq & (s[j], \quad\quad rank[j+1]) & s_j \in C^1 \\ (s[i..i+1], rank[i+2]) & \leq & (s[j..j+1], rank[j+2]) & s_j \in C^2 \end{cases}$$

   Vergewissern Sie sich, dass C lexikographisch sortiert ist und damit das Suffixarray induziert.

Notation:
- Alle Indizes fangen bei 0 an – analog zu Kapitel 9.3.6, auf dem die Aufgabe basiert.
- $s[i\ (..j)\ ]$: Zeichen an Stelle i (bis j) in s (z.B. $s[1..3] =$ `ber`)
- s_i: Suffix von s ab Stelle i (z.B. $s_2 =$ `erakadabera`)

Aufgabe 5 (*Rechnen+Analyse: LCP-Array*)

Gegeben sei die Zeichenkette $s =$ `salsadipp`.

a) Geben Sie das Suffixarray für s an.

b) Geben Sie das LCP-Array für s an.

Im Folgenden sei ein String t sowie dessen Suffixarray SA[·] und dessen LCP-Array LCP[·] gegeben.

c) Wie kann der längste sich wiederholende Substring in t effizient bestimmt werden?
(der Substring darf sich dabei selbst überlappen)

d) Wie viele paarweise unterschiedliche Substrings kann ein String der Länge n maximal besitzen? Wie kann die tatsächliche Anzahl für einen konkreten String t bestimmt werden?

e) Ein String lässt sich schlecht komprimieren, wenn er wenig Redundanz besitzt. Ein Maß dafür ist die Anzahl paarweise unterschiedlicher Substrings normiert auf die mögliche Gesamtanzahl unterschiedlicher Substrings. Geben Sie an, wie dieses Maß für t berechnet werden kann.

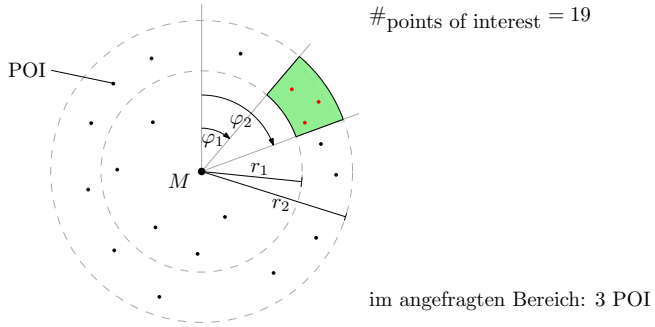
Aufgabe 6 (*Kleinaufgaben: Geometrie-Entwurf*)

Entwerfen Sie einen Algorithmus, der ...

a) in Zeit $O(n \log n)$ ein geschlossenes, kreuzungsfreies Polygon aus n Punkten $P \in \mathbb{R}^2$ konstruiert.

b) in Zeit $O(n)$ für eine Menge von n Filialen einen Standort für ein Zentrallager berechnet, so dass der maximale Abstand (in Luftlinie) zwischen Zentrallager und allen Filialen minimiert wird.

c) (*) in Zeit $O(\log n)$ die Anzahl an *points of interest* (POI) um einen fixen Mittelpunkt M in einem Winkelbereich $[\varphi_1, \varphi_2]$ und einem Entfernungsbereich $[r_1, r_2]$ bestimmt.



Bei n POI ist eine Vorverabeitungszeit von $O(n \log n)$ und ein Platzverbrauch von $O(n)$ erlaubt.

Aufgabe 7 (*Analyse+Entwurf: Überdeckungsproblem*)

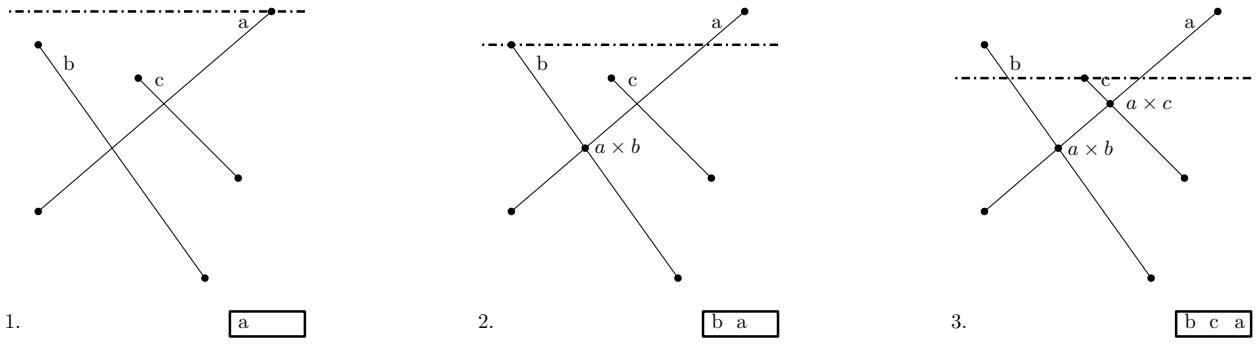
Zur Gebietsüberwachung wurde in einem weitläufigen Gelände ein Sensornetz aus mehreren Millionen Knoten ausgelegt. Die Positionsdaten der Knoten wurden per Funkübertragung an einer zentralen Stelle gesammelt. Jeder Sensorknoten überwacht ein kreisförmiges Gebiet mit Radius r. Durch Fehler in der Ausbringung können dabei starke Überlappungen der von den Knoten überwachten Gebiete entstanden sein.
Um diese Information zu einem späteren Zeitpunkt ggf. nutzen zu können, haben die Betreiber beschlossen, dass alle Knotenpaare bestimmt werden sollen, deren Gebiete sich teilweise überlappen.

a) Zeigen Sie, dass ein *Sweepline*-Algorithmus, der einfach alle (im Rahmen der Algorithmenausführung) aktiven Sensorknoten auf Überlappung prüft, in quadratischer Laufzeit resultieren kann, auch wenn die Ausgabekomplexität (Anzahl überlappender Knotenpaare) linear ist.

b) Überlegen Sie, wie Sie dennoch einen *Sweepline*-Algorithmus verwenden können, um das Problem in Linearzeit zu lösen, falls die Ausgabekomplexität linear ist.

Aufgabe 8 (*Entwurf: Platzeffizienter Linienschnitt*)

In der Vorlesung wurde ein *Sweepline*-Algorithmus zur Bestimmung der Schnitte zwischen n Liniensegmenten behandelt. Der Algorithmus arbeitet eine Liste an Ereignispunkten (Schnittpunkte, Linienanfänge und Linienenden), in Form einer nach der y-Position sortierten *Queue*, ab. Gleichzeitig führt er eine Liste aktiver Kanten in sortierter Reihenfolge.
Das betrachtete Problem lässt sich in seiner Laufzeitkomplexität nie besser lösen als durch die Anzahl Schnittpunkte vorgegeben. Liegen z.B. $O(n^2)$ Schnittpunkte vor, so kann bestenfalls eine quadratische Laufzeitkomplexität erreicht werden.
Für den Algorithmus aus der Vorlesung gilt die gleiche Einschränkung auch für den Platzbedarf. Die *Queue* muss bis zu $O(n+k)$ Ereignispunkte gleichzeitig halten, bei insgesamt k Linienschnitten. Dies liegt unter anderem daran, dass einmal erkannte Schnittpunkte auch zwischen Liniensegmenten existieren können, die in der sortierten Liste nicht (mehr) benachbart sind. Dies sei durch folgendes Beispiel illustriert:



Im Beispiel wird zuerst der Schnittpunkt $a \times b$ zwischen den Linien a und b bestimmt. Nach der Aktivierung von Linie c ist der Schnittpunkt weiterhin in der *Queue*, die Linien a und b sind allerdings nicht mehr benachbart.

Modifizieren Sie den Algorithmus so, dass er nur noch $O(n)$ Platz für die Ereignispunkte benötigt.

Aufgabe 9 (*Analyse+Entwurf:* Graham's Scan)

Betrachten Sie den *Graham's Scan* Algorithmus zur Bestimmung der konvexen Hülle einer Punktemenge $P \in \mathbb{R}^2$ mit $|P| = n$. In der Vorlesung wurde eine lexikographische Sortierung der Punkte vorgeschlagen, mit der Folge dass die obere und untere Hülle getrennt berechnet werden mussten.

a) Geben Sie eine geeignetere Sortierung der Punkte an, so dass die gesamte konvexe Hülle in einem Durchlauf berechnet werden kann.

b) Zeigen Sie, dass der *Graham's Scan* Algorithmus nicht für jede beliebige Sortierung der Punkte eine korrekte konvexe Hülle liefert (Randfälle ausgenommen).

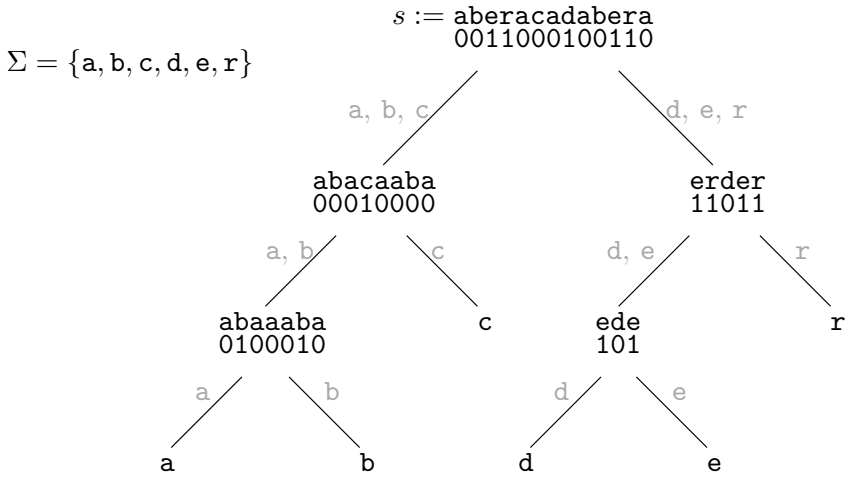
c) Zeigen Sie anhand eines Beispiels, dass es möglich ist, dass der *Graham's Scan* Algorithmus p schon zur Hülle hinzugefügte Punkte hintereinander verwirft für beliebig großes p.

d) Eine Schneidemaschine bringt Stoffe anhand eines Schnittmusters in die gewünschte Form. Ein Schnittmuster ist dabei durch ein Polygon aus n Ecken definiert. Die Schneidemaschine kann beliebige konvexe Stoffe bearbeiten.

   Entwerfen Sie einen Algorithmus, der den minimal möglichen Verschnitt für ein gegebenes Stoffmuster in Linearzeit berechnet. Sie können davon ausgehen, dass die Ecken des Polygons als geschlossener Kantenzug vorliegen.

Aufgabe 10 (*Entwurf:* Wavelet Trees *für Zeichenketten (*)*)

Aus der Vorlesung ist Ihnen eine schnelle `rank(i)` Operation auf Bitfolgen bekannt. diese Operation berechnet die Anzahl Einsen in der Bitfolge bis zu Position i. Dies ist in konstanter Zeit möglich. Für schnelle Suchalgorithmen auf Zeichenketten benötigt man eine Erweiterung dieser Operation, `rank(c,i)`, die die Anzahl an Zeichen `c` bis Position i liefert.
Für diese (und weitere) Aufgaben sind *Wavelet Trees* für Zeichenketten ein sehr nützliches Werkzeug. Im folgenden soll diese Datenstruktur kurz vorgestellt werden: Für eine Zeichenkette s über dem Alphabet Σ wird ein *Wavelet Tree* wie in folgendem Bild aufgebaut.



Die Blätter des Baumes halten die verwendeten Zeichen des Alphabets, die inneren Knoten halten Bitvektoren, die die Menge der Zeichen partitionieren. Eine 0 bzw. 1 gibt an, ob das Zeichen an dieser Stelle zur unteren (aufgerundet) oder zur oberen (abgerundet) Hälfte des in diesem Knoten aktiven Teil des Alphabets gehört. In der Wurzel ist das gesamte Alphabet aktiv. Dieses wird in jedem Nachfolger halbiert (die Kantenbeschriftungen geben das aktive Alphabet an). Die angegebenen Zeichenketten über den Bitvektoren dienen nur der Anschauung, im *Wavelet Tree* werden sie nicht gespeichert.

Gehen Sie davon aus, dass die Bitvektoren eine $\mathtt{rank}_{0/1}\mathtt{(i)}$ Operation zur Bestimmung der Nullen bzw. Einsen bis Position i in $O(1)$ besitzen. Außerdem können Sie annehmen, dass *Wavelet Trees* balanciert sind. Entwerfen Sie unter diesen Voraussetzungen einen Algorithmus, der ...

a) `access(s,i)` berechnet, d.h. der das i-te Zeichen der Zeichenkette s zurückliefert.
(Bsp.: `access(s,4) = 'r'`)

b) `rank(s,c,i)` berechnet, d.h. der die Anzahl an Zeichen c in s bis Position i angibt.
(Bsp.: `rank(s,'a',7) = 3`)

Beide Algorithmen dürfen $O(\log u)$ Zeit benötigen, mit $u = \min(|s|, |\Sigma|)$.

Ausgabe: 24.01.2012
Abgabe: keine Abgabe, keine Korrektur