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

Musterlösungen

Aufgabe 1 (*Rechnen: Kompression*)

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

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

Musterlösung:

a) Die komprimierte Zeichenkette lautet: $c = 1, 2, 4, 4, 3, 7, 3$
Das aufgebaute Wörterbuch `D` lautet:
$D = \{(1, \texttt{a}), (2, \texttt{b}), (3, \texttt{c}), (4, \texttt{ab}), (5, \texttt{ba}), (6, \texttt{aba}), (7, \texttt{abc}), (8, \texttt{ca}), (9, \texttt{abcc})\}$

Index	Zeichenfolge
1	a
2	b
3	c
4	ab
5	ba
6	aba
7	abc
8	ca
9	abcc

b) Die dekomprimierte Zeichenkette lautet: $s = \texttt{a}, \texttt{b}, \texttt{ab}, \texttt{aba}, \texttt{ba}, \texttt{b}$
Das aufgebaute Wörterbuch D lautet:
$D = \{(1, \texttt{a}), (2, \texttt{b}), (3, \texttt{ab}), (4, \texttt{ba}), (5, \texttt{aba}), (6, \texttt{abab}), (7, \texttt{bab})\}$

Index	Zeichenfolge
1	a
2	b
3	ab
4	ba
5	aba
6	abab
7	bab

c) Die übermittelte Zeichenkette lautet $c = 1, \underline{3}, 2, \underline{4}, 5$.
Das übermittelte Wörterbuch D lautet:
$D = \{(1, \underline{\texttt{a}}), (2, \texttt{b}), (3, \underline{\texttt{aa}}), (4, \texttt{aab}), (5, \underline{\texttt{ba}}), (6, \texttt{aabb})\}$

Die dekomprimierte Zeichenkette lautet $s = \texttt{a}, \texttt{aa}, \texttt{b}, \texttt{aab}, \texttt{ba}$.

Lösungsweg:
$D[1] = \texttt{a}$, da das Wörterbuch zunächst das komplette Alphabet enthalten muss. $D[3] = \texttt{aa}$, da der vierte Eintrag des Wörterbuchs bereits drei Zeichen enthält und sich jeder neue Eintrag aus einem vorherigen Eintrag (hier: `aa`) und einem weiteren Zeichen (hier: `b`) zusammensetzt. $s[2] = \texttt{aa}$, da $s[3] = \texttt{b}$ und man dies für $D[4]$ benötigt. $s[4]$ in dekodierter Form muss mit einem `a` beginnen, sonst wäre $D[5] = \texttt{bb}$ und $D[6]$ müsste auch mit `b` beginnen. Damit ist $D[4] = \texttt{ba}$. Um schließlich noch $D[6] = \texttt{aabb}$ erzeugen zu können, muss $s[4] = \texttt{aab}$ sein (wie bei $D[4]$).

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: border[10] = 3, border[11] = 2, border[12] = 1).

Musterlösung:

a) Es ergibt sich folgendes *border-array*:

p	a	b	a	c	a	b	a	b	c
border[]	-1	0	0	1	0	1	2	3	2

b) Das Suchmuster wird insgesamt 4 mal angelegt:

t	a	b	a	c	a	b	a	b	b	a	b	a	c	a	b	a	b	c	
	a	b	a	c	a	b	a	b	c										Stelle 1
							a	b	a	c	a	b	a	b	c				Stelle 7
									a	b	a	c	a	b	a	b	c		Stelle 9
										a	b	a	c	a	b	a	b	c	Stelle 10

c) Das Muster muss maximal $|t| - |p| + 1$ mal angelegt werden. Dieser Fall tritt z.B. auf, falls das erste Zeichen von p nicht in t auftaucht. Das Muster muss minimal $\lceil |t|/(|p| - 1) \rceil$ mal angelegt werden. Dieser Fall tritt z.B. auf, falls p aus einer Folge paarweise unterschiedlicher Zeichen besteht und t aus Konkatenationen von p ohne das letzte Zeichen.

d) Zu zeigen oder widerlegen ist die Existenz von drei Einträgen:

border$[i] = k$, border$[i+1] = k-1$, border$[i+2] = k-2$.

Einträge dieser Art können nicht existieren, da in diesem Fall gelten würde:

für border$[i+0] = k-0$: $p_{k-2} = p_{i-3}$, $p_{k-1} = p_{i-2}$, $p_k = p_{i-1}$,
für border$[i+1] = k-1$: $p_{k-2} = p_{i-1}$, $p_{k-1} = p_i$ und
für border$[i+2] = k-2$: $p_{k-2} = p_{i+1}$.

Damit, würde sich ergeben:

$p_{k-2} = p_{i-3} = p_{i-1} = p_{i+1} = p_k$,
$p_{k-1} = p_{i-2} = p_i$.

In diesem Fall wäre aber border$[i+2] = k$, da
$p_{k-2} = p_{i-1} = p_{i-3}$, $p_{k-1} = p_i = p_{i-2}$ und $p_k = p_{i+1} = p_{i-1}$.
Andernfalls wäre auch border$[i+0] \neq k$.

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

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

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.

Musterlösung:

a) Die Entropie ist definiert als $H(s) = -\sum_i \log_2 p(s[i])$ mit $p(c) = |\{s[i] : s[i] = c\}|/n$ gleich der relativen Häufigkeit, dass Zeichen c in Zeichenkette s auftaucht. Es ergibt sich:

$$\begin{aligned} H(s) &= -\left(\log_2 \frac{5}{10} + \log_2 \frac{5}{10} + \log_2 \frac{5}{10} + \dots\right) \\ &= -\left(10 \cdot \log_2 \frac{1}{2}\right) \\ &= 10 \end{aligned}$$

b) Vorgehen:

- bilde zyklische Permutationen von s (Endzeichen \$ nicht vergessen!)
- sortiere die Permutationen (\$ ist kleiner als alle anderen Zeichen!)
- lese das Ergebnis s^{BWT} aus der letzten Spalte ab

Ausführung:

```
ababababab$       $ababababab
bababababa$       ab$abababab
abababab$ab       abab$ababab
bababab$aba       ababab$abab
ababab$abab       abababab$ab
babab$ababa   →   ababababab$   → s^BWT = bbbbb$aaaaa
abab$ababab       b$ababababa
bab$abababa       bab$abababa
ab$ababababab     babab$ababa
b$ababababa       bababab$aba
$ababababab       bababababa$
```

Da die *Burrows-Wheeler-Transformation* lediglich die Zeichen der Zeichenkette permutiert, ändert sich die Entropie nicht.

c) Die kodierte Zeichenkette zu s^{BWT} lautet: $c^{BWT} = 2, 1, 1, 1, 1, 2, 1, 1, 1, 1$
Sie baut sich wie folgt auf:

Index i	Y	$s^{BWT}[i]$	$c^{BWT}[i]$
1	ab	b	2
2	ba	b	1
3	ba	b	1
4	ba	b	1
5	ba	b	1
6	ba	a	2
7	ab	a	1
8	ab	a	1
9	ab	a	1
10	ab	a	1

Entropie von c^{BWT}:

$$\begin{aligned} H(c) &= -\left(\log_2 \frac{2}{10} + \log_2 \frac{8}{10} + \log_2 \frac{8}{10} + \log_2 \frac{8}{10} + \log_2 \frac{8}{10} + \right. \\ &\qquad \left. \log_2 \frac{2}{10} + \log_2 \frac{8}{10} + \log_2 \frac{8}{10} + \log_2 \frac{8}{10} + \log_2 \frac{8}{10}\right) \\ &= -\left(2 \cdot \log_2 \frac{1}{5} + 8 \cdot \log_2 \frac{4}{5}\right) \\ &\approx 7.22 \end{aligned}$$

Musterlösung:

d) Die kodierte Zeichenkette zu s lautet: $c = 1, 2, 2, 2, 2, 2, 2, 2, 2, 2$
Sie baut sich wie folgt auf:

Index i	Y	$s^{BWT}[i]$	$c^{BWT}[i]$
1	ab	a	1
2	ab	b	2
3	ba	a	2
4	ab	b	2
5	ba	a	2
6	ab	b	2
7	ba	a	2
8	ab	b	2
9	ba	a	2
10	ab	b	2

Entropie von c^{BWT}:

$$\begin{aligned} H(c) &= -\Big(\log_2\frac{1}{10} + \log_2\frac{9}{10} + \log_2\frac{9}{10} + \log_2\frac{9}{10} + \log_2\frac{9}{10} + \\ &\qquad \log_2\frac{9}{10} + \log_2\frac{9}{10} + \log_2\frac{9}{10} + \log_2\frac{9}{10}\Big) \\ &= -\Big(1\cdot\log_2\frac{1}{10} + 9\cdot\log_2\frac{9}{10}\Big) \\ &\approx 4.69 \end{aligned}$$

Erstaunlicherweise ist die *Move-to-Front* Kodierung von die Zeichenkette s erfolgreicher als von Zeichenkette s^{BWT} nach *Burrows-Wheeler-Transformation*. Dies kann bei einigen Spezialfällen auftreten. Eine anschließende Komprimierung von c (z.B. mit *Huffmann-Kodierung*) würde c stärker verkleinern können als c^{BWT}.

Zum Vergleich wird auf der nächsten Seite unter Teilaufgabe x) als weiteres Verfahren eine *Lempl-Ziv* Kompression von s und s^{BWT} ausgeführt und deren Entropie betrachtet.

e) Vorgehen:

- schreibe s^{BWT} in Spaltenform auf
- sortiere zeilenweise
- schreibe s^{BWT} in Spaltenform vor die bisherigen Spalten
- wiederhole bis $|s^{BWT}|$ Spalten sortiert wurden
- die oberste Zeile gibt s an

Ausführung:

```
b       $       b$      $a      b$a     $ab     b$ab
c       a       ca      ab      cab     ab$     cab$
$ sort  a  add  $a  sort ab  add $ab  sort abc  add $abc
a  ---> b  ---> ab  ---> b$  ---> ab$ ---> b$a  ---> ab$a
a       b       ab      bc      abc     bca     abca
b       c       bc      ca      bca     cab     bcab
```

```
      $abc      b$abc     $abca     b$abca    $abcab
      ab$a      cab$a     ab$ab     cab$ab    ab$abc
sort  abca  add $abca sort abcab add $abcab sort abcab$
---> b$ab ---> ab$ab ---> b$abc ---> ab$abc ---> b$abca
      bcab      abcab     bcab$     abcab$    ccab$a
      cab$      bcab$     cab$a     ccab$a    cab$ab
```

Die gesuchte Zeichenfolge lautet $s =$ `abcab`.

Musterlösung:

x) *zusätzliche Betrachtungen: Kompression von s und s^{BWT} mit dem* Lempel-Ziv *Verfahren* (war nicht Teil der Aufgabenstellung)

Die komprimierte Zeichenkette zu s lautet: $c = 1, 2, 3, 5, 4, 2$
Das aufgebaute Wörterbuch D lautet:
$D = \{(1, \texttt{a}), (2, \texttt{b}), (3, \texttt{ab}), (4, \texttt{ba}), (5, \texttt{aba}), (6, \texttt{abab}), (7, \texttt{bab})\}$

Index	Zeichenfolge
1	a
2	b
3	ab
4	ba
5	aba
6	abab
7	bab

Entropie von c:

$$\begin{aligned} H(c) &= -\left(\log_2 \frac{1}{6} + \log_2 \frac{1}{3} + \log_2 \frac{1}{6} + \log_2 \frac{1}{6} + \log_2 \frac{1}{6} + \log_2 \frac{1}{3}\right) \\ &= -\left(4 \cdot \log_2 \frac{1}{6} + 2 \cdot \log_2 \frac{1}{3}\right) \\ &\approx 13.5 \end{aligned}$$

Die komprimierte Zeichenkette zu s^{BWT} lautet: $c^{BWT} = 2, 3, 3, 1, 6, 6$
Das aufgebaute Wörterbuch D^{BWT} lautet:
$D^{BWT} = \{(1, \texttt{a}), (2, \texttt{b}), (3, \texttt{bb}), (4, \texttt{bbb}), (5, \texttt{bba}), (6, \texttt{aa}), (7, \texttt{aaa})\}$

Index	Zeichenfolge
1	a
2	b
3	bb
4	bbb
5	bba
6	aa
7	aaa

Entropie von c^{BWT}:

$$\begin{aligned} H(c^{BWT}) &= -\left(\log_2 \frac{1}{6} + \log_2 \frac{1}{3} + \log_2 \frac{1}{3} + \log_2 \frac{1}{6} + \log_2 \frac{1}{3} + \log_2 \frac{1}{3}\right) \\ &= -\left(2 \cdot \log_2 \frac{1}{6} + 4 \cdot \log_2 \frac{1}{3}\right) \\ &\approx 11.5 \end{aligned}$$

Wie man sieht, ist die Entropie von c^{BWT} kleiner als die Entropie von c. Damit konnte der String s^{BWT} nach *Burrows-Wheeler-Transformation* besser komprimiert werden als davor. Allerdings ist die die Entropie beider komprimierter Strings größer ist als die des unkomprimierten Strings! Dieser Effekt tritt bei kurzen Strings häufig auf, da sie nicht genug Redundanz bieten.

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:

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

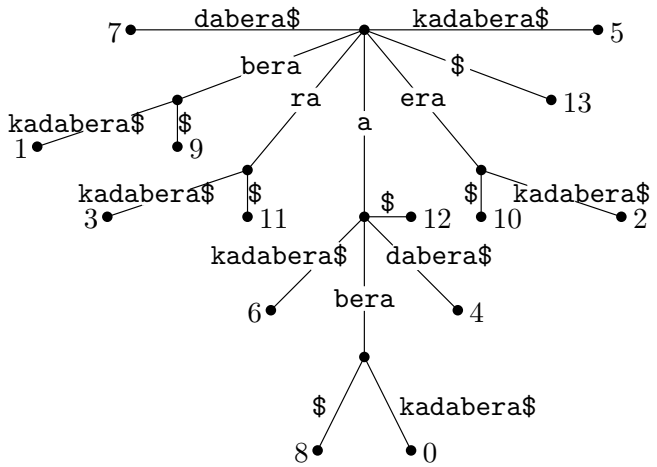
   Vergewissern Sie sich, dass C lexikographisch sortiert ist und damit das Suffixarray induziert.

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`)

Musterlösung:
Als Referenz zunächst Zeichenkette s mit ihren Indizes (beachten Sie das Abschlusszeichen `$`!):

Index i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$s[i]$	a	b	e	r	a	k	a	d	a	b	e	r	a	$

a) Der Suffixbaum für s als kompaktierter Trie:



Jeder Weg von der Wurzel zu einem Blatt gibt einen Suffix von s an. Der Wert am Blatt gibt den Index des Suffix an (d.h. die Stelle in der Zeichenkette an der der Suffix beginnt).

b) Das Suffixarray SA für s lautet:
SA $= \langle 13, 12, 8, 0, 6, 4, 9, 1, 7, 10, 2, 5, 11, 3 \rangle$.

Index i	SA$[i]$	$s_{\mathrm{SA}[i]}$
0	13	`$`
1	12	`a$`
2	8	`abera$`
3	0	`aberakadabera$`
4	6	`adabera$`
5	4	`akadabera$`
6	9	`bera$`
7	1	`berakadabera$`
8	7	`dabera$`
9	10	`era$`
10	2	`erakadabera$`
11	5	`kadabera$`
12	11	`ra$`
13	3	`rakadabera$`

In der rechten Spalte der Suffixtabelle sind die durch das Suffixarray indizierten Suffixe aufgetragen. Man sieht, dass sie durch das Suffixarray in alphabetischer Reihenfolge geordnet vorliegen.

Musterlösung:

c) Die Tripelsequenzen lauten:

$R^1 = \langle \texttt{ber}, \texttt{aka}, \texttt{dab}, \texttt{era}, \texttt{\$\$\$} \rangle$
$R^2 = \langle \texttt{era}, \texttt{kad}, \texttt{abe}, \texttt{ra\$} \rangle$

Beachten Sie, dass das letzte Tripel von R^1 mit weiteren Abschlusszeichen \$ ergänzt wurde.

d) Die sortierten Tripel von R^{12} ohne Duplikate ergeben folgende Ränge:

Index i	4	7	1	0	2	3, 5	6	8
$R^{12}[i]$	\$\$\$	abe	aka	ber	dab	era	kad	ra\$
Rang	0	1	2	3	4	5	6	7

e) Die Ränge definieren eine eindeutige Bezeichnung der Tripel von R^{12}. Damit kann R^{12} dargestellt werden als $s^{12} = \langle 3, 2, 4, 5, 0, 5, 6, 1, 7 \rangle$.

Die Sequenz s^{12} enthält das selbe Zeichen (5) mehrfach. Ansonsten wäre über die Ränge bereits eine vollständige Sortierung von s^{12} –und damit auch von C^{12}– bestimmt. Um diese Sortierung zu erhalten, bestimmt man rekursiv mit dem DC3-Algorithmus das Suffixarray für s^{12}.

f) Das Suffixarray für R^{12} ist –nach Konstruktion– gleich dem Suffixarray für s^{12}. Es ergibt sich zu $\mathrm{SA}^{12} = \langle 9, 4, 7, 1, 0, 2, 3, 5, 6, 8 \rangle$.

Index i	$\mathrm{SA}^{12}[i]$	$s^{12}_{\mathrm{SA}^{12}[i]}$	$R^{12}_{\mathrm{SA}^{12}[i]}$
0	9	$\langle . \rangle$	$\langle \rangle$
1	4	$\langle 0, 5, 6, 1, 7, . \rangle$	$\langle \texttt{\$\$\$}, \texttt{era}, \texttt{kad}, \texttt{abe}, \texttt{ra\$}, \rangle$
2	7	$\langle 1, 7, . \rangle$	$\langle \texttt{abe}, \texttt{ra\$}, \rangle$
3	1	$\langle 2, 4, 5, 0, 5, 6, 1, 7, . \rangle$	$\langle \texttt{aka}, \texttt{dab}, \texttt{era}, \texttt{\$\$\$}, \texttt{era}, \texttt{kad}, \texttt{abe}, \texttt{ra\$}, \rangle$
4	0	$\langle 3, 2, 4, 5, 0, 5, 6, 1, 7, . \rangle$	$\langle \texttt{ber}, \texttt{aka}, \texttt{dab}, \texttt{era}, \texttt{\$\$\$}, \texttt{era}, \texttt{kad}, \texttt{abe}, \texttt{ra\$}, \rangle$
5	2	$\langle 4, 5, 0, 5, 6, 1, 7, . \rangle$	$\langle \texttt{dab}, \texttt{era}, \texttt{\$\$\$}, \texttt{era}, \texttt{kad}, \texttt{abe}, \texttt{ra\$}, \rangle$
6	3	$\langle 5, 0, 5, 6, 1, 7, . \rangle$	$\langle \texttt{era}, \texttt{\$\$\$}, \texttt{era}, \texttt{kad}, \texttt{abe}, \texttt{ra\$}, \rangle$
7	5	$\langle 5, 6, 1, 7, . \rangle$	$\langle \texttt{era}, \texttt{kad}, \texttt{abe}, \texttt{ra\$}, \rangle$
8	6	$\langle 6, 1, 7, . \rangle$	$\langle \texttt{kad}, \texttt{abe}, \texttt{ra\$}, \rangle$
9	8	$\langle 7, . \rangle$	$\langle \texttt{ra\$}, \rangle$

Beachten Sie das zusätzliche Abschlusszeichen . für s^{12}. Es ist funktional identisch zum \$ für s. Zur leichteren Unterscheidung wurde aber ein anderes Zeichen gewählt. Das Abschlusszeichen benötigt keine Entsprechung in R^{12}, daher ist hier nur ein leeres Tripel eingetragen. Es wird für die Bestimmung von SA^{12} benötigt, kann aber im weiteren Verlauf ignoriert werden.

Das Suffixarray gibt eine Sortierung der Elemente von R^{12} bzw. s^{12} und damit auch von C^{12} an (für C^{12} ist dies spätestens dann ersichtlich, wenn man in der rechten Spalte alle Tripel nach \$\$\$ entfernt und die restlichen in einer Zeile konkateniert).

g) Aus dem Suffixarray lässt sich folgende Rang-Funktion ableiten:

Index i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
$s[i]$	a	b	e	r	a	k	a	d	a	b	e	r	a	\$	\$	...
$rank[i]$	$\perp$	4	7	$\perp$	3	8	$\perp$	5	2	$\perp$	6	9	$\perp$	1	0	...

Beachten Sie, dass $rank[i] = \perp$ anstatt 0 gesetzt wurde f.a. $(i \bmod 3) = 0$. Dies ist zulässig, da diese Einträge von *rank* nie verwendet werden.

h) Es ergeben sich die folgenden Tupel (für $s_i \in C^0$ gilt $(i \bmod 3) = 0$, $i < |s|$):

Index i	0	3	6	9	12
$(s[i], rank[i+1])$	(a, 4)	(r, 3)	(a, 5)	(b, 9)	(a, 1)

Sortiert ergibt sich: $\langle (\texttt{a}, 1), (\texttt{a}, 4), (\texttt{a}, 5), (\texttt{b}, 9), (\texttt{r}, 3) \rangle$. Diese Sortierung entspricht einer Sortierung von C^0. Alle unterschiedlichen Anfangsbuchstaben der Suffixe sind korrekt sortiert. Bei gleichem Anfangsbuchstaben ist über *rank* eine korrekte Sortierung ab dem zweiten Zeichen gegeben.

Musterlösung:

i) Für das Mischen werden C^0 und C^{12} in sortierter Reihenfolge benötigt. Diese Reihenfolge wurde in den vorherigen Teilaufgaben berechnet. Für den Vergleich beim Mischen wird zusätzlich für jedes Suffix eine bestimmte Tupeldarstellung benötigt.

Für C^0 ergeben sich die sortierte Reihenfolge und die benötigten Tupel wie folgt:

Index i	s_i	$(s[i], rank[i+1])$	$(s[i..i+1], rank[i+2])$
12	a$ $\in C^0$	(a, 1)	(a$, 0)
0	aberakadabera$ $\in C^0$	(a, 4)	(ab, 7)
6	adabera$ $\in C^0$	(a, 5)	(ad, 2)
9	bera$ $\in C^0$	(b, 6)	(be, 9)
3	rakadabera$ $\in C^0$	(r, 3)	(ra, 8)

Für C^{12} sind die sortierte Reihenfolge (z.B. aus $rank$ abzulesen) und die benötigten Tupel:

Index i	s_i	$(s[i], rank[i+1])$	$(s[i..i+1], rank[i+2])$
13	$ $\in C^1$	($, 0)	
8	abera$ $\in C^2$		(ab, 6)
4	akadabera$ $\in C^1$	(a, 8)	
1	berakadabera$ $\in C^1$	(b, 7)	
7	dabera$ $\in C^1$	(d, 2)	
10	era$ $\in C^1$	(e, 9)	
2	erakadabera$ $\in C^2$		(er, 3)
5	kadabera$ $\in C^2$		(ka, 5)
11	ra$ $\in C^2$		(ra, 1)

Zusammengemischt ergibt sich C zu:

Index i	s_i	$(s[i], rank[i+1])$	$(s[i..i+1], rank[i+2])$
13	$ $\in C^1$	($, 0)	
12	a$ $\in C^0$	(a, 1)	(a$, 0)
8	abera$ $\in C^2$		(ab, 6)
0	aberakadabera$ $\in C^0$	(a, 4)	(ab, 7)
6	adabera$ $\in C^0$	(a, 5)	(ad, 2)
4	akadabera$ $\in C^1$	(a, 8)	
9	bera$ $\in C^0$	(b, 6)	(be, 9)
1	berakadabera$ $\in C^1$	(b, 7)	
7	dabera$ $\in C^1$	(d, 2)	
10	era$ $\in C^1$	(e, 9)	
2	erakadabera$ $\in C^2$		(er, 3)
5	kadabera$ $\in C^2$		(ka, 5)
11	ra$ $\in C^2$		(ra, 1)
3	rakadabera$ $\in C^0$	(r, 3)	(ra, 8)

Die Suffixe in C sind offensichtlich lexikographisch sortiert (zweite Spalte). Damit erhält man das Suffixarray für s als SA $= \langle 13, 12, 8, 0, 6, 4, 9, 1, 7, 10, 2, 5, 11, 3 \rangle$.

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.

Musterlösung:

a) Das Suffixarray SA für s lautet:
SA $= \langle 10, 5, 2, 6, 7, 3, 9, 8, 4, 1 \rangle$.

Index i	SA$[i]$	$s_{\text{SA}[i]}$
1	10	`$`
2	5	`adipp$`
3	2	`alsadipp$`
4	6	`dipp$`
5	7	`ipp$`
6	3	`lsadipp$`
7	9	`p$`
8	8	`pp$`
9	4	`sadipp$`
10	1	`salsadipp$`

In der rechten Spalte der Suffixtabelle sind die durch das Suffixarray indizierten Suffixe aufgetragen. Man sieht, dass sie durch das Suffixarray in alphabetischer Reihenfolge geordnet vorliegen.

b) Das LCP-Array LCP für s lautet:
LCP $= \langle 0, 0, 1, 0, 0, 0, 0, 1, 0, 2 \rangle$.

Der Eintrag LCP$[i]$ gibt die Länge des gemeinsamen Präfixes von $s_{\text{SA}[i-1]}$ und $s_{\text{SA}[i]}$ an. Damit ist LCP[1] nicht definiert. Nach Konvention setzt man normalerweise LCP$[1] = 0$.

c) Das Suffixarray SA$[\cdot]$ enthält alle Suffixe von T lexikographisch sortiert. Damit sind die Einträge mit dem längstem gemeinsamen Präfix –und damit mit dem längsten Substring– benachbart. Da jeder Eintrag LCP$[i]$ im LCP-Array die Länge des gemeinsamen Präfixes von benachbarten Suffixen $s_{\text{SA}[i-1]}$ und $s_{\text{SA}[i]}$ angibt, genügt es, das Maximum von LCP$[\cdot]$ zu bestimmen, um die Position des längsten sich wiederholenden Substring zu erhalten.

d) Die Menge aller Substrings eines Strings ist durch die Menge aller Präfixe seiner Suffixe gegeben. In einem String mit maximal vielen unterschiedlichen Substrings besitzen keine zwei Suffixe ein gemeinsames Präfix. Damit steuert jedes Suffix s genau $|s|$ zur Menge der paarweise unterschiedlichen Substrings bei. Somit ist die gesuchte Anzahl $\sum_{i=1}^{n} |s_{SA[i]}| = \sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$.

Die Anzahl paarweiser unterschiedlicher Substrings für einen konkreten String t ist gegeben durch $\#_{pds} = \sum_{i=1}^{n} |s_{\text{SA}[i]}| - \text{LCP}[i]$. Dies folgt aus folgender Überlegung:
Sei LCP$[j] = k$. Dies bedeutet, dass $s_{\text{SA}[i-1]}$ und $s_{\text{SA}[i]}$ ein gemeinsames Präfix der Länge k besitzen. Das wiederum heißt, dass die k Suffixe dieses gemeinsamen Präfix nicht gezählt werden dürfen, da sie gemeinsam und nicht paarweise verschieden sind.

Die Länge eines Suffix lässt sich mit Hilfe des Suffix-Arrays bestimmen zu $|s_{\text{SA}[i]}| = n - SA[i]$.

e) Dies lässt sich direkt aus der vorherigen Teilaufgabe ableiten: $\frac{2}{n \cdot (n+1)} \sum_{i=1}^{n} P[i]$.

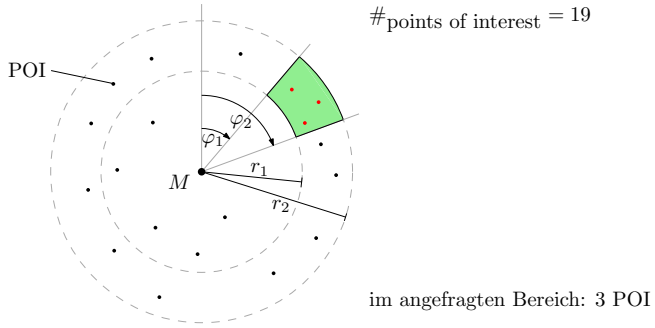
Aufgabe 6 (*Kleinaufgaben: Geometrie-Entwurf*)

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.

Musterlösung:

a) Bestimme den Mittelpunkt M aller Punkte aus P in $O(n)$. Sortiere die Punkte in $O(n \log n)$ im Uhrzeigersinn um M. Bei gleichem Winkel hat der Punkt, der näher am Mittelpunkt ist Vorrang. Füge die Punkte in dieser Reihenfolge zu einem Polygon zusammen.

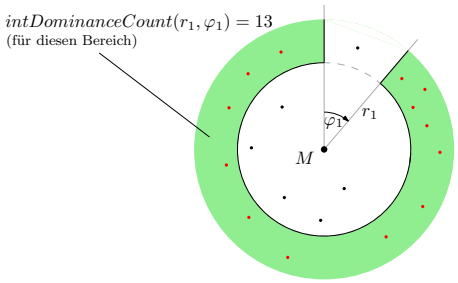
Das erzeugte Polygon ist offensichtlich geschlossen, wenn auch der letzte Punkt mit dem ersten Punkt verbunden wird. Das Polygon ist außerdem kreuzungsfrei, da es monoton in einer Richtung aufgebaut wird (im Uhrzeigersinn und von innen nach nach außen).

b) Das Problem kann mit dem Algorithmus zur Bestimmung der kleinsten einschließenden Kugel gelöst werden. Der Mittelpunkt der berechneten Kugel (bzw. des Kreises in 2D) minimiert den maximalen Abstand zu allen Filialen und ist der gesuchte Standort für das Zentrallager.

c) Die Anfrage kann durch *Wavelet Trees* mit den geforderten Eigenschaften gelöst werden. An Stelle von karthesischen Koordinaten (x, y) werden Kreiskoordinaten (r, φ) verwendet, um den *Wavelet Tree* aufzubauen. In diesem Fall gibt eine Anfrage $intDominanceCount(r_1, \varphi_1)$ die Anzahl an POI im Bereich $[r_1, \infty)$ und $[\phi_1, 2\pi)$ an (siehe Bild). Damit kann die gewünschte Bereichsanfrage konstruiert werden:

$$\begin{aligned} intRangeCount(r_1, r_2, \varphi_1, \varphi_2) \quad &= intDominanceCount(r_2, \varphi_1) - intDominanceCount(r_2, \varphi_2) \\ &\quad - intDominanceCount(r_1, \varphi_1) + intDominanceCount(r_1, \varphi_2) \end{aligned}$$

Sollte der Winkelbereich die 0° überstreichen, teilt man die Anfrage in zwei getrennte Anfragen mit den Winkelbereichen $[\varphi_1, 2\pi)$ bzw. $[0, \varphi_2)$, deren Ergebnisse man addiert.



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.

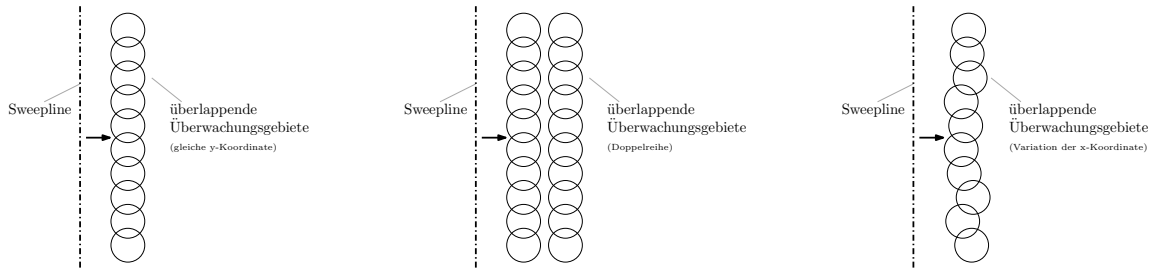
a) Zeigen Sie, dass ein *Sweepline*-Algorithmus, der einfach alle (im Rahmen der Algorithmenausführung) aktiven Sensorknoten auf Überlappung prüft, in quadratischer Laufzeit resultieren kann, auch wenn die Ausgabekomplexität (Anzahl überlappender Knotenpaare) linear ist.

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

Musterlösung:

a) Viele *Sweepline*-Algorithmen sind für eine effiziente Ausführung darauf angewissen, dass die Zahl aktiver Elemente gering ist im Vergleich zur Anzahl vorhandener Elemente. Dies kann bei dem gestellten Problem allerdings nicht garantiert werden.

Betrachte eine Sortierung der Knoten nach x-Koordinate und anschließend nach y-Koordinate. Der *Sweepline*-Algorithmus durchlaufe die Knoten in dieser Sortierung. Sollten durch eine ungünstige Verteilung alle Sensorknoten auf der selben y-Koordinate liegen, so sind zu einem Zeitpunkt alle Knoten gleichzeitig aktiv und man muss jeden Knoten mit jedem vergleichen. Dieser Fall resultiert somit in quadratischer Laufzeit, obwohl nur linear viele Schnitte auftreten. Er könnte dennoch effizient gelöst werden, da die Knoten auch in der y-Koordinate geordnet sind und deshalb in sortierter Reihenfolge aktiviert werden. Allerdings kann diese Ordnung durch Aufteilung der Elemente in zwei gleich lange Gruppen oder Variation der x-Koordinate in Schritten von r/n zerstört werden. Die folgenden Abbildungen illustrieren die drei Fälle:

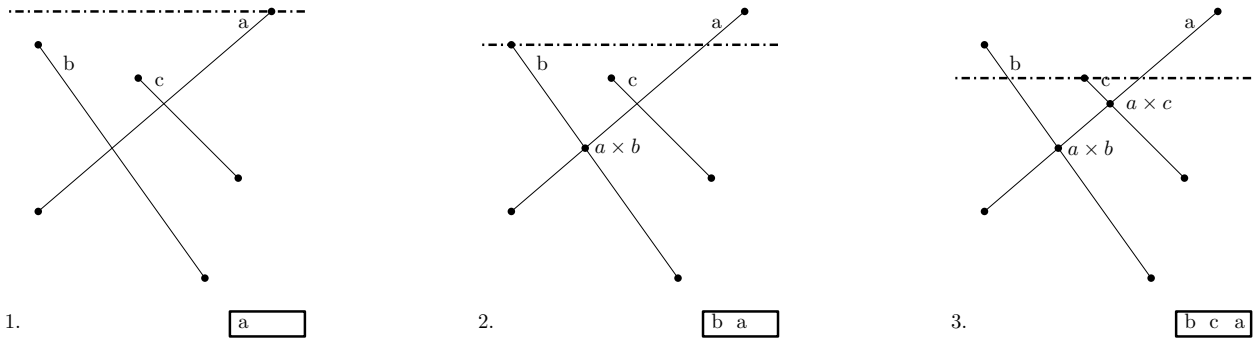


b) Wie in der vorherigen Teilaufgabe gezeigt, besteht das Problem darin, dass gleichzeitig viele Knoten aktiv sein können. Ein Trick zur Umgehung dieses Problems ist die Verwendung von Buckets. Verwaltet man die Menge aller aktiven Elemente in einer sortieren Liste an Buckets (augeteilt anhand der y-Koordinate, Bucketgröße $> r$), so sind nur Vergleiche mit den Elementen aus dem eigenen und den zwei benachbarten Buckets nötig. Da nur reine Überlagerungstests durchgeführt werden, sind bei geeigneter Wahl der Bucketgröße sogar nur Vergleiche mit den benachbarten Buckets nötig, da sich die Kreise eines Buckets garantiert überlagern (gilt für Bucketgrößen $< 2r$).

Sollten sich alle Elemente auf benachbarte Buckets verteilen, lässt sich auch mit diesem Trick eine quadratische Menge an Tests nicht vermeiden. Durch geschickte Wahl der Bucketgrößen lässt sich die Wahrscheinlichkeit eines solchen Falles allerdings meist so weit reduzieren, dass quadratische Laufzeit nur auftritt, wenn die Ausgabekomplexität quadratisch in der Eingabegröße ist. Eine solche Komplexität lässt sich dann allerdings nicht vermeiden.

Aufgabe 8 (*Entwurf: Platzeffizienter Linienschnitt*)

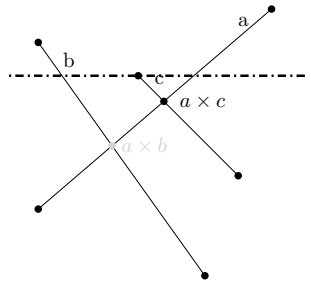
In der Vorlesung wurde ein *Sweepline*-Algorithmus zur Bestimmung der Schnitte zwischen n Liniensegmenten behandelt. Der Algorithmus arbeitet eine Liste an Ereignispunkten (Schnittpunkte, Linienanfänge und Linienenden), in Form einer nach der y-Position sortierten *Queue*, ab. Gleichzeitig führt er eine 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:



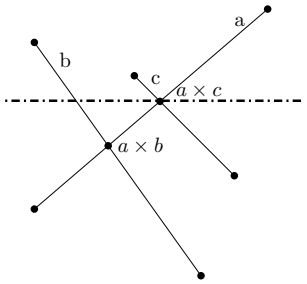
Im Beispiel wird zuerst der Schnittpunkt $a \times b$ zwischen den Linien a und b bestimmt. Nach der Aktivierung von Linie c ist der Schnittpunkt weiterhin in der *Queue*, die Linien a und b sind allerdings nicht mehr benachbart.

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

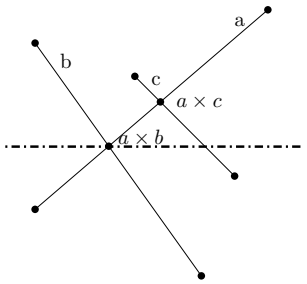
Musterlösung:
Die Lösung beruht auf der Tatsache, dass die an einem Schnittpunkt beteiligten Linien unmittelbar vor der Bearbeitung des Schnittpunktes in der Liste aktiver Kanten benachbart sein müssen. Damit können Schnittpunkte zeitweise verworfen werden, die zu momentan nicht mehr benachbarten Linien gehören. Dies sei durch folgendes Beispiel illustriert:



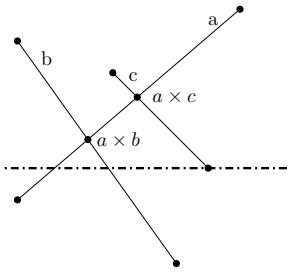
1. b c a



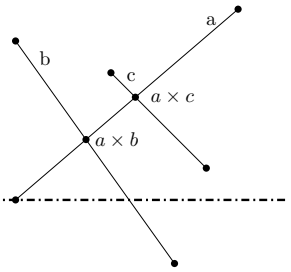
2. b a c



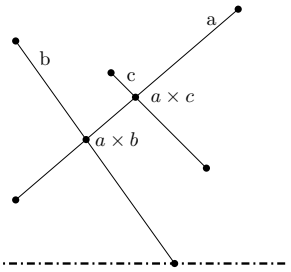
3. b a c



4. a b



5. b



6.

Folgt man dem Beispiel, so kann bei Aktivierung von Line c Schnittpunkt $a \times b$ verworfen werden. Erst bei Abarbeitung von Schnittpunkt $a \times c$ wird er wieder eingefügt.
Die benötigte Überprüfung ist im Allgemeinen sowieso nötig und hat somit keinen Einfluss auf die Laufzeit des Algorithmus. Durch diese Änderung kann sich zwischen jeweils zwei aktiven Linien nur jeweils ein aktiver Schnittpunkt in der *Queue* befinden. Da es maximal $n - 1$ benachbarte aktive Linienpaare geben kann, enthält die *Queue* insgesamt nur die geforderten $O(n)$ Ereignispunkte.

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.

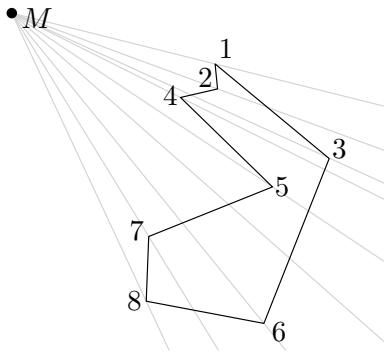
d) Eine Schneidemaschine bringt Stoffe anhand eines Schnittmusters in die gewünschte Form. Ein Schnittmuster ist dabei durch ein Polygon aus n Ecken definiert. Die Schneidemaschine kann beliebige konvexe Stoffe bearbeiten.

   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.

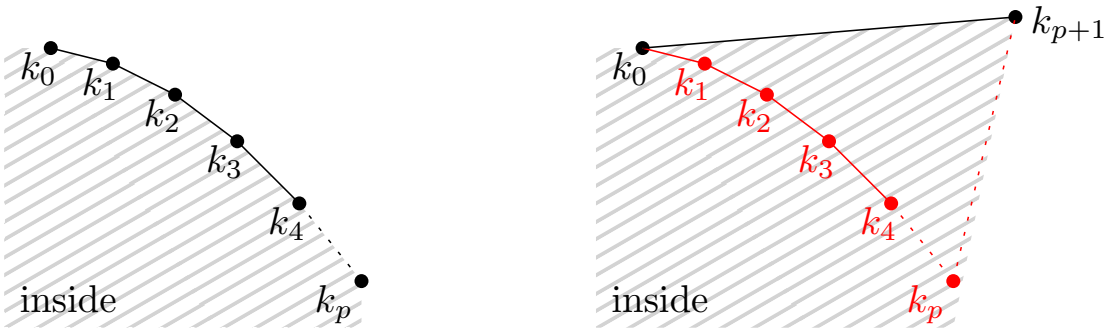
Musterlösung:

a) Eine zirkuläre Sortierung der Punkte um einen Mittelpunkt innerhalb der konvexen Hülle erfüllt diese Anforderung. Als Startpunkt für *Graham's Scan* wählt man einen Punkt der sicher auf der Hülle liegt, z.B. den Punkt mit kleinster x Koordinate. Zu berücksichtigende Sonderfälle treten auf, wenn sich Punkte in der gleichen Richtung vom Mittelpunkt aus gesehen befinden. Hier müssen die inneren vor den äußeren Punkten abgearbeitet werden.

b) Betrachte eine zirkuläre Sortierung der Punkte um einen Mittelpunkt außerhalb der konvexen Hülle. Wie im folgenden Bild zu sehen, springt die Reihenfolge der Punkte wild umher, wenn sie zirkulär um M sortiert werden.

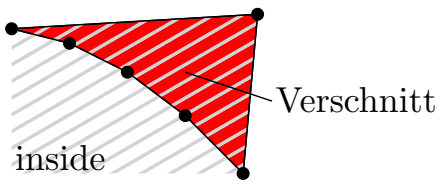


c) Das geforderte Verhalten tritt auf, wenn beim Scan für p Punkte hintereinander eine 'Rechtsdrehung' stattfindet und anschließend eine 'Linksdrehung', so dass der $p+1$-te Punkt über dem ersten der Reihe liegt. Folgendes Bild veranschaulicht dies.



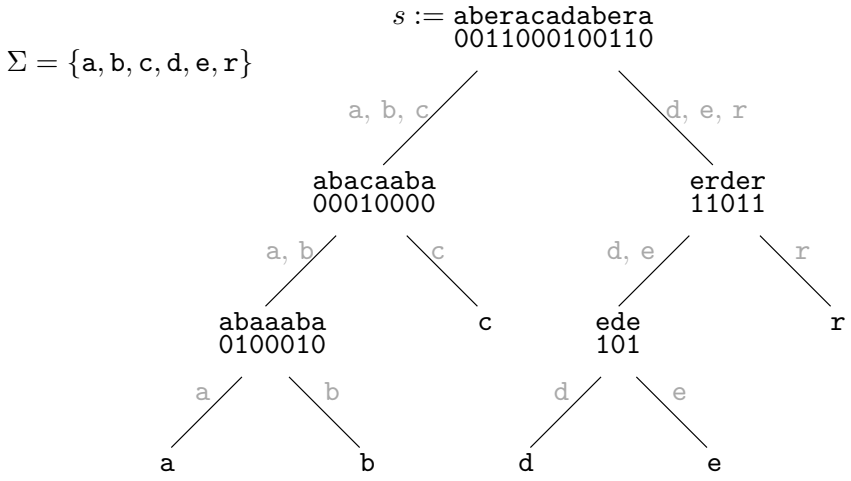
Links ist der Zustand nach Scan des p-ten Punktes zu sehen, rechts der Zustand nach Scan des $p+1$-ten Punktes. Die Punkte k_1 bis k_p wurden aus der vorläufigen konvexen Hülle entfernt.

d) Der geforderte Algorithmus ist ein modifizierter *Graham's Scan*. Die Ecken des Schnittmusters bilden die Eingabepunkte für den Algorithmus. Da sie schon sortiert vorliegen, entfällt der teuerste Schritt von *Graham' Scan*. Der Rest arbeitet in Linearzeit. Der Verschnitt wird bestimmt, indem jedes Mal, wenn eine 'Linksdrehung' beim Scan auftritt, die zusätzliche Fläche (im Bild rot) berechnet und über den gesamten Lauf des Algorithmus aufsummiert wird.



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 $\texttt{rank}_{0/1}\texttt{(i)}$ 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 $u = \min(|s|, |\Sigma|)$.

Musterlösung:
Ein balancierter *Wavelet Tree* hat offensichtlich eine Höhe von $O(\log u)$. Jeder Knoten v speichert eine Bitfolge B_v sowie einen Zeiger zu seinem linken und rechten Nachfolger v_l, v_r (für v innerer Knoten) bzw. ein Zeichen $label_v$ (für v Blattknoten).
Als Eingabe für beide Operationen wird die Zeichenkette in *Wavelet Tree* Darstelleung verwendet, repräsentiert durch ihre Wurzel. Beide Algorithmen führen einen Abstieg im *Wavelet Tree* von der Wurzel bis zu einem Blatt durch. Der Baum hat eine Höhe von $O(\log u)$, in jedem Knoten wird $O(1)$ Arbeit verrichtet. Damit ergibt sich ein Aufwand von $O(\log u)$ für beide Algorithmen.

a)
```
function ACCESS(v : Wavelet Tree, i : Integer)
    if v is a leaf then
        return label_v
    else if B_v[i] = 0 then
        return ACCESS(v_l, rank_0(B_v, i))
    else
        return ACCESS(v_r, rank_1(B_v, i))
    end if
end function
```

Es wird ein rekursiver Abstieg im *Wavelet Tree* durchgeführt und mit jedem Aufruf die gesuchte Position i an die noch vorhandene Zeichenmenge angepasst – $rank_0(B_v, i)$ für Abstieg nach links; $rank_1(B_v, i)$ für Abstieg nach rechts.

b)
```
function RANK(v : Wavelet Tree, c : Character, i : Integer)
    if v is a leaf then
        if label_v == c then
            return i
        else
            return ⊥
        end if
    else if c ∈ labels(v_l) then
        return RANK(v_l, c, rank_0(B_v, i))
    else
        return RANK(v_r, c, rank_1(B_v, i))
    end if
end function
```

Der Algorithmus funktioniert analog zu der oberen Teilaufgabe, außer dass die Richtung des Abstiegs nicht über den Wert des Bitvektors an Position i festgelegt wird, sondern in welcher Richtung das gesuchte Zeichen c liegt. Die dafür nötige Abfrage, ob c zum aktiven Alphabet des linken Nachfolgers gehört, $c \in labels(v_l)$, kann in konstanter Zeit beantwortet werden: Zu Beginn wird ein Index auf das erste und letzte Zeichen des Alphabets gespeichert, der mit jedem rekursiven Aufruf angepasst wird.