Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Moritz Laupichler
Hans-Peter Lehmann

6. Übungsblatt zu Algorithmen II im WS 2022/2023

http://algo2.iti.kit.edu/AlgorithmenII_WS22.php
{sanders, moritz.laupichler, hans-peter.lehmann}@kit.edu

Musterlösungen

Aufgabe 1 (*Analyse: ADAC Mitgliedschaft*)

Der "**A**utomobil **D**urch **A**lgorithmiker **C**lub" (ADAC) leistet auf Autobahnen Pannenhilfe. Ein Autofahrer hat in seiner Zeit als Verkehrsteilnehmer n Pannen, $n \in \mathbb{N}_{\geq 0}$, für die er die Hilfe des ADAC in Anspruch nehmen muss. Für jede geleistete Pannenhilfe verlangt der Club eine Aufwandsentschädigung abhängig von der Schwere der Panne. Mitglieder beim ADAC müssen lediglich ein Viertel dieser Kosten bezahlen. Eine lebenslange Mitgliedschaft kann man sich durch eine Einmalzahlung in Höhe von 1000 DM (**D**ijkstra **M**ark) sichern.
Da nicht schon mit Erwerb des Führerscheins klar ist, wie viele Pannen man in seinem Leben haben wird und wie schwerwiegend diese sein werden, stellt sich die Frage, ab wann es sich lohnt eine Mitgliedschaft beim ADAC zu erwerben.

a) Geben Sie eine Strategie an, die einen kompetitiven Faktor (competitive ratio) von ∞ erreicht. Begründen Sie kurz.

b) Wie gut ist die Strategie, sich nie eine Mitgliedschaft beim ADAC zu sichern? Begründen Sie.

c) Zeigen Sie, dass folgende Strategie einen kompetitiven Faktor $c = 3$ hat. Die Strategie ist, sich beim Pannenhelfer eine Mitgliedschaft zu kaufen, wenn die momentan von ihm bearbeitete Panne die Gesamtausgaben für Pannen (ohne Mitgliedschaft) auf über 500 DM anheben würde.

Hinweis: Verwenden Sie die summierten Gesamtkosten K über alle Pannen (ohne Mitgliederrabatt).

Musterlösung:
Ein Algorithmus ALG wird als *streng c-kompetitiv* bezeichnet, wenn für alle Eingaben I gilt

$$c = \sup_I \frac{ALG(I)}{OPT(I)}$$

Dieser Wert wird als *kompetitiver Faktor* (*competitive ratio*) bezeichnet.

Der Übersichtlichkeit halber wird im Folgenden ohne Einheiten gerechnet:

a) Wenn man sofort mit Erhalt der Fahrerlaubnis eine Mitgliedschaft beim ADAC erwirbt, gibt man im schlimmsten Fall 1000 DM aus, nimmt aber die Hilfe des ADAC nie in Anspruch. Dies ergibt einen kompetitiven Faktor von $c = \frac{1000}{0} = \infty$.

b) Wenn man sich nie eine Mitgliedschaft kauft, gibt man offensichtlich höchstens 4 mal soviel für den ADAC aus wie ein Mitglied. Die summierten Gesamtkosten für alle Pannen seien mit K bezeichnet. Für $K \to \infty$ konvergiert der kompetitive Faktor gegen $c = \frac{K}{1000+K/4} \to 4$.

c) Die summierten Gesamtkosten für alle Pannen seien mit K und die summierten Kosten vor Beitritt zum ADAC mit $K_1 \leq 500$ bezeichnet. Die eigene Strategie liefert

$$ALG = \begin{cases} K_1 + 1000 + (K - K_1)/4 & K > 500, \\ K & \text{sonst} \end{cases}$$

in Abhängigkeit davon, ob man jemals über 500 DM Kosten für Pannen hat oder nicht. Die optimale Strategie ist durch

$$OPT = \begin{cases} 1000 + K/4 & K \geq 1333\frac{1}{3}, \\ K & \text{sonst} \end{cases}$$

gegeben. Entweder kauft man sich sofort eine Mitgliedschaft oder nie. Die Grenzkosten ergeben sich durch Lösen von $1000 + K/4 \stackrel{!}{=} K$. Der kompetitive Fakor ist der maximale Quotient von ALG und OPT. Allgemein gilt

$$\frac{ALG(K)}{OPT(K)} = \begin{cases} \frac{K}{K} & K \leq 500, \\ \frac{1375+K/4}{1000+K/4} & K \geq 1333\frac{1}{3}, \\ \frac{1375+K/4}{K} & \text{sonst} \end{cases}$$

(mit $K_1 = 500$ gesetzt, da wir nur am maximalen Wert interessiert sind). Das Supremum dieses Quotienten ist 3 für $K \to 500, K > 500$. Damit ist der kompetitive Faktor dieser Strategie

$$c = \sup_K \frac{ALG(K)}{OPT(K)} = 3.$$

Aufgabe 2 (*Analyse: Online-gaming-Algorithmen*)

Angestellte in einem Rechenzentrum haben einen recht eintönigen Job. Ihnen stehen zwar die größten Rechner zur Verfügung, Sie dürfen diese aber nicht selbst verwenden. Stattdessen heißt es nur, die Maschinen möglichst gut auszulasten und Rechnungen zu schreiben. Ein ziemlich langweiliger Job möchte man meinen – zum Glück gibt es noch Computerspiele.
Ein Mitarbeiter hat zu Weihnachten ein neues Computerspiel erhalten, dass er liebsten andauernd spielen würde. Diesem Wunsch steht leider seine Arbeit im Wege. Eine neue Dienstanweisung besagt, dass die teuren Großrechner zu mindestens 50% ausgelastet sein müssen. Da das Scheduling in diesem Rechenzentren noch von Hand durchgeführt wird, muss der spielfreudige Mitarbeiter die laufenden Jobs überwachen und schnell neue Jobs auf leerlaufende Maschinen verteilen. So bleibt leider nur wenig Zeit zum Spielen am Arbeitsplatz.
Jeder Job hat eine Mindestlaufzeit von 5 Minuten. Die Zeit zum Verteilen der Jobs kann für die Bestimmung der Auslastung vernachlässigt werden. Der Mitarbeiter kann in dieser Zeit aber nicht spielen. Ein Job hat während seiner Ausführung die Maschine exklusiv. Es gibt keine automatischen Benachrichtigungen über das Ende eines Jobs. Der Mitarbeiter muss dies selbst periodisch überprüfen.

a) Entwerfen Sie einen *online scheduling* Algorithmus, der die freie Zeit des Mitarbeiters unter Einhaltung der Nebenbedingungen maximiert, wenn er einen Großrechner zu betreuen hat.

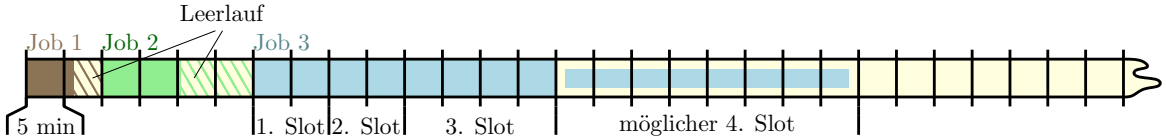
b) Zeigen Sie, dass Ihr Algorithmus optimal bzgl. der Freizeit des Mitarbeiters ist.

Musterlösung:

a) Wir definieren den Algorithmus induktiv. Durch die Mindestlaufzeit von 5 Minuten kann man jedem Job zunächst ein Zeitslot von 10 Minuten zuweisen. Nach Ablauf dieses Zeitslots muss der Mitarbeiter nachschauen, ob der Job bereits abgeschlossen ist, in der Zwischenzeit kann er spielen. Falls der Job in dieser Zeit beendet wurde, stand die Maschine höchstens 50% der Zeit still bis der Mitarbeiter dies erkannt und einen neuen Job gestartet hat. Andernfalls muss ein neuer Zeitslot für den noch laufenden Job zugewiesen werden.

Die Länge des neuen Zeitslots wird gleich der bisherigen Gesamtlaufzeit des Jobs (10 Minuten) gewählt. Dies verdoppelt die mögliche Gesamtlaufzeit des Jobs bis zur nächsten Überprüfung durch den Mitarbeiter auf 20 Minuten. Dadurch wird sichergestellt, dass der Rechner maximal die Hälfte der Zeit leerläuft. Wenn der Job ε Zeiteinheiten nach Start des neuen Zeitslots endet, ist er insgesamt $10+\varepsilon$ Minuten gelaufen und der Rechner damit zu $\frac{10+\varepsilon}{20} > 50\%$ ausgelastet gewesen. Sollte der neue Zeitslot auch nicht genügen, wird dieses Vorgehen wiederholt, bis der Job endet (neuer Zeitslot von $20, 40, \ldots$ Minuten, maximale Gesamtlaufzeit von $40, 80, \ldots$ Minuten).

Folgende Abbildung verdeutlicht den Ablauf des Algorithmus:

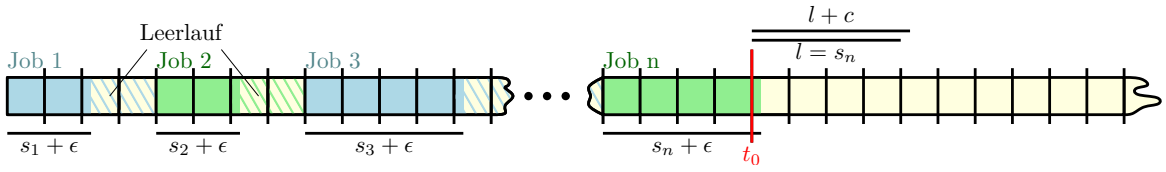


Kein Job erhält mehr als das Doppelte seiner Laufzeit an Zeitslots zugeteilt. Somit ist der Großrechner im Durchschnitt zu mindestens 50% ausgelastet – wie gefordert.

Auch wenn die Aufgabenstellung einen eher scherzhaften Hintergund hat, so ist die vorgestellte Technik der Verdopplung (*doubling*) ein nützliches Hilfsmittel beim Entwurf von *online* Algorithmen. Insbesondere Algorithmen die Suchschritte benötigen, können durch derartige Techniken oft schon recht gute Approximationsgarantien (bzw. kompetitive Faktoren) liefern.

Musterlösung:

b) Angenommen, es existiere ein Algorithmus, der längere Zeitslots zwischen zwei Überprüfungen erlaubt als unser Algorithmus. Dann existiert für eine Folge an Jobs ein Zeitpunkt t_0, an dem die zugeteilten Zeitslots beider Algorithmen erstmals voneinander abweichen. Unser Algorithmus weise einen Zeitslot der Länge l zu, der potentiell bessere Algorithmus einen Zeitslot der Länge $l' = l + c$. Betrachte konkret eine Folge an Jobs, für die unser Algorithmus genau die minimale Auslastungsgrenze von 50% einhält (siehe Grafik).



Job i benötigt Zeit $s_i + \epsilon$ mit s_i gleich einer Dauer, nach der eine Überprüfung stattfindet – also nach $10, 20, 40, 80, \ldots$ Minuten. Zum Zeitpunkt t_0 weist unser Algorithmus einen Zeitslot von $l = s_n$ als Verlängerung zu, der andere Algorithmus einen Zeitslot von $l' = l + c = s_n + c$. Wähle $\epsilon < \frac{c}{2n}$, so ergibt sich nach Abarbeitung dieses Zeitslots eine Auslastung von

$$\frac{\sum_{i=1}^{n} s_i + \epsilon}{c + \sum_{i=1}^{n} 2s_i} = \frac{n \cdot \epsilon + \sum_{i=1}^{n} s_i}{c + 2 \cdot \sum_{i=1}^{n} s_i} < \frac{\frac{c}{2} + \sum_{i=1}^{n} s_i}{2 \cdot \left(\frac{c}{2} + \cdot \sum_{i=1}^{n} s_i\right)} = \frac{1}{2}$$

für den alternativen Algorithmus. Beachte: Startet zu dem Zeitpunkt t_0 ein neuer Job, so ist das Ergebnis analog zu erreichen durch eine Wahl von $s_n = 10$, der doppelten Mindestlaufzeit des Algorithmus. Damit hält er nicht – wie gefordert – eine minimale Auslastung von 50% ein. Unser Algorithmus verwendet also bereits die maximal möglichen Zeitslots zwischen zwei Überprüfungen und ermöglicht dem Mitarbeiter die meiste Freizeit.

Aufgabe 3 (*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, falls der erste von diesen drei Einträgen auf ein Suffix der Größe $k \geq 3$ verweist, welches gleichzeitig echtes Präfix ist.

Beispiel: $\text{border}[10] = 3,\ \text{border}[11] = 2,\ \text{border}[12] = 1$

Kein Beispiel: $\text{border}[10] = 2,\ \text{border}[11] = 1,\ \text{border}[12] = 0$

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:

$\text{border}[i] = k,\ \text{border}[i+1] = k-1, \text{border}[i+2] = k-2.$

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

für $\text{border}[i+0] = k-0$: $P_{k-2} = P_{i-3},\ P_{k-1} = P_{i-2},\ P_k = P_{i-1}$,
für $\text{border}[i+1] = k-1$: $P_{k-2} = P_{i-1},\ P_{k-1} = P_i$ und
für $\text{border}[i+2] = k-2$: $P_{k-2} = P_{i+1}$.

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 $\text{border}[i+2] = k$, 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 $\text{border}[i+0] \neq k$.

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:

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

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:

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

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

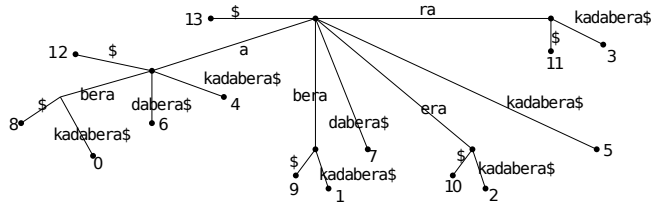
Notation:
- Alle Indizes fangen bei 0 an – analog zu Kapitel 9.3.6, auf dem die Aufgabe basiert.

- $s[i\ (..j)\ ]$: 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, 6)	(a, 1)

Sortiert ergibt sich: $\langle (\texttt{a}, 1), (\texttt{a}, 4), (\texttt{a}, 5), (\texttt{b}, 6), (\texttt{r}, 3) \rangle$. 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 den Suffixbaum für s an.

b) Geben Sie das Suffixarray für s an.

c) Geben Sie das LCP-Array für s an.

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

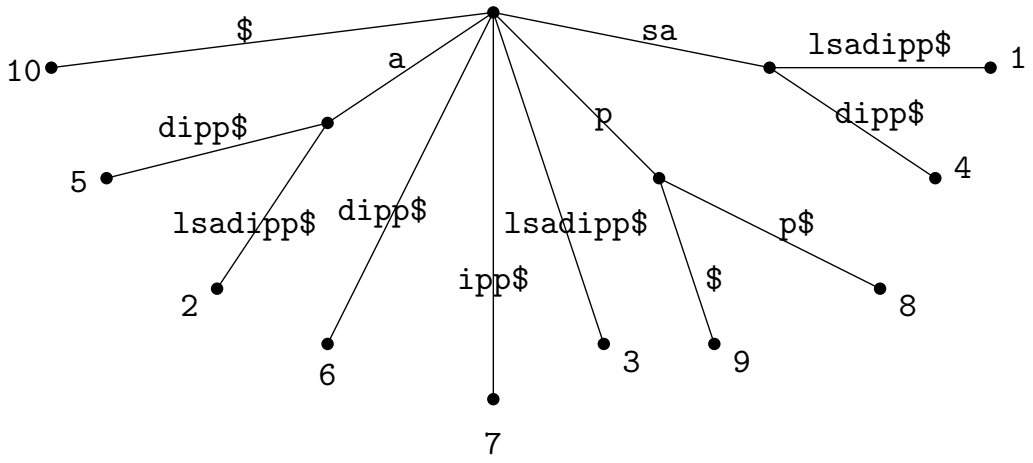
d) Wie kann der längste sich wiederholende Substring in T effizient bestimmt werden?
(der Substring darf sich dabei selbst überlappen)

e) 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?

f) 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) 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 $= \langle 10, 5, 2, 6, 7, 3, 9, 8, 4, 1 \rangle$.

Index i	SA$[i]$	$s_{\mathrm{SA}[i]}$
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.

c) 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 $s_{\mathrm{SA}[i-1]}$ und $s_{\mathrm{SA}[i]}$ an. Damit ist LCP[1] nicht definiert. Nach Konvention setzt man normalerweise LCP$[1] = 0$.

Musterlösung:

d) Das Suffixarray SA[·] 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 $s_{\mathrm{SA}[i-1]}$ und $s_{\mathrm{SA}[i]}$ angibt, genügt es, das Maximum von LCP[·] zu bestimmen, um die Position des längsten sich wiederholenden Substring zu erhalten.

e) 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 $\#_{pds} = \sum_{i=1}^{n} |s_{\mathrm{SA}[i]}| - \mathrm{LCP}[i]$. Dies folgt aus folgender Überlegung:
   Sei LCP[j] $= k$. Dies bedeutet, dass $s_{\mathrm{SA}[i-1]}$ und $s_{\mathrm{SA}[i]}$ 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 $|s_{\mathrm{SA}[i]}| = n - SA[i]$.

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

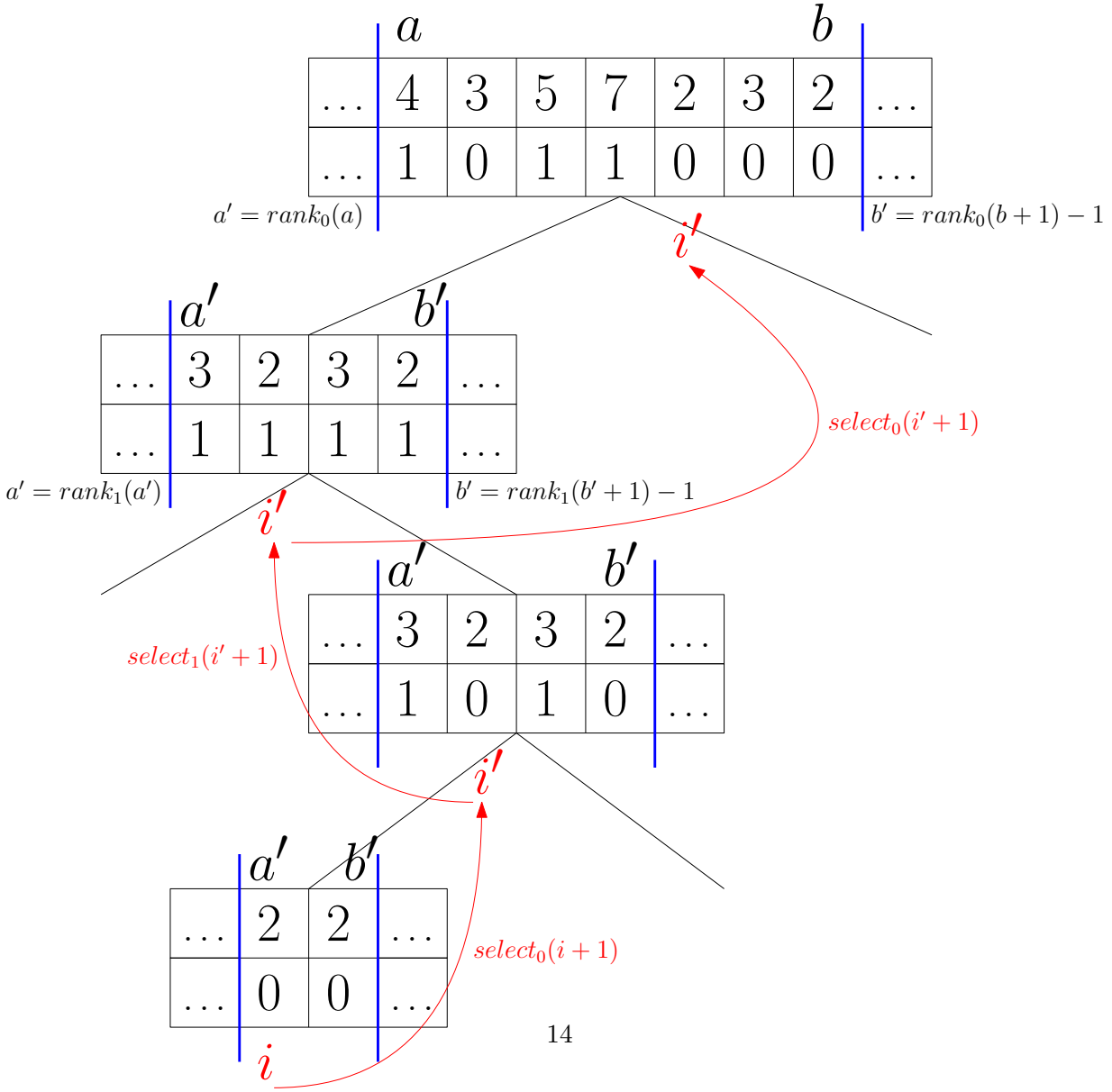
Aufgabe 6 (*RMQ in Wavelet Trees*)

Gegeben sei ein Universum von Zahlen $\mathcal{U}$ und ein Feld A von Zahlen aus $\mathcal{U}$. Geben sie einen Algorithmus an, mit dem sich unter Benutzung eines Wavelet Trees in $\log |\mathcal{U}|$ Zeit $\arg\min_i \{A[i] \mid i \in [a, b]\}$ für Parameter a, b berechnen lässt.

Musterlösung:
Sei WT der Wavelet Tree für unser Feld A.

1: **function** RMQ(WT, a, b)
2: **if** WT ist Blatt **then**
3: **return** a
4: **end if**
5: $a' \leftarrow rank_0(a)$
6: $b' \leftarrow rank_0(b+1) - 1$
7: **if** $b' - a' \geq 0$ **then** ▷ Minimum ist in linkem Teilbaum
8: $i' \leftarrow \mathsf{RMQ}(WT \rightarrow child_{left}, a', b')$
9: $i \leftarrow select_0(i'+1)$
10: **else** ▷ Minimum ist in rechtem Teilbaum
11: $i' \leftarrow \mathsf{RMQ}(WT \rightarrow child_{right}, a - a', b - b' - 1)$ ▷ $rank_1(x) = x - rank_0(x)$
12: $i \leftarrow select_1(i'+1)$
13: **end if**
14: **return** i
15: **end function**

Dieser Algorithmus ist leicht modifizierbar um das linkeste, rechteste oder mittlere Minimum auszugeben.



Aufgabe 7 (*Rechnen: Kompression*)

a) Bestimmen Sie die *Lempel-Ziv-Faktoren* $f_i = (l_i, p_i)$ des Textes $T_1 =$ `abbbababbbb$`.

b) Gegeben seien folgende *Lempel-Ziv-Faktoren*. Bestimmen Sie den Text T_2, aus dem sie entstanden sind.

$$(0, \texttt{a}), (5, 1), (0, \texttt{b}), (6, 1), (7, 7), (0, \texttt{\$})$$

c) Angenommen, jedes Zeichen kann in 1 byte, sowie jeder *Lempel-Ziv-Faktor* in 2 byte repräsentiert werden. Wie viel Prozent des Speicherplatzes von T_2 konnte somit eingespart werden?

Musterlösung:

a) Die Lempel-Ziv-Faktoren lauten:

f_1	`a`	(0, `a`)
f_2	`b`	(0, `b`)
f_3	`bb`	(2, 2)
f_4	`ab`	(2, 1)
f_5	`abbb`	(4, 1)
f_6	`b`	(1, 2)
f_7	`$`	(0, `$`)

b) $T_2 =$ `aaaaaabaaaaaabaaaaaa$`

f_1	`a`	(0, `a`)
f_2	`aaaaa`	(5, 1)
f_3	`b`	(0, `b`)
f_4	`aaaaaa`	(6, 1)
f_5	`baaaaaa`	(7, 7)
f_6	`$`	(0, `$`)

c) $|T_1| = 21$ benötigt 21 byte. $|f_i| = 6$ benötigt 12 byte. Dadurch ergibt sich eine Einsparung von $1 - 12/21 \approx 42\%$.