Nachname:

Vorname:

Matrikelnummer:

**Lösungsvorschlag**

Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. P. Sanders 16.03.2023

# Klausur Algorithmen II

| | | |
|---|---|---|
| Aufgabe 1. | Kleinaufgaben | 9 Punkte |
| Aufgabe 2. | Priority Queues | 11 Punkte |
| Aufgabe 3. | Externe Algorithmen: Wissenschaftliche Papers | 12 Punkte |
| Aufgabe 4. | Parametrisierte Algorithmen: Konjunktive Normalform | 7 Punkte |
| Aufgabe 5. | Stringology: Suffix-Array- und LCP-Array-Konstruktion | 10 Punkte |
| Aufgabe 6. | Geometrische Algorithmen: “Quadro-Bot” | 11 Punkte |

Bitte beachten Sie:

- Als Hilfsmittel ist nur **ein** DIN-A4 Blatt mit Ihren **handschriftlichen** Notizen zugelassen.
- Schreiben Sie Ihre Antworten nur in blau oder schwarz, mit dokumentenechtem Stift.
- **Schreiben** Sie auf **alle** Blätter der Klausur und Zusatzblätter Ihre **Matrikelnummer**.
- Die Klausur enthält **17 Seiten**.
- Zum Bestehen der Klausur sind 30 Punkte hinreichend.

Matrikelnummer:

EK

ZK



Lösungsvorschlag

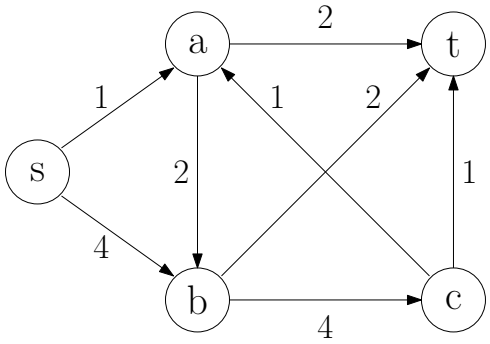
**Aufgabe 1.** Kleinaufgaben [9 Punkte]

**a.** Sie möchten $p = 2^d$ Rechner zu einem Hyperwürfel-Verbindungsnetz verbinden. Wie viele Kabel, die jeweils zwei Rechner verbinden, müssen Sie kaufen? Was ist die Entfernung (Anzahl Kabel) zwischen den am weitesten entfernten Rechnern? [2 Punkte]

## Lösung

Anzahl Kanten eines Hyperwürfels: $\frac{p \log p}{2} = \frac{d2^d}{2}$.
Durchmesser: $d = \log(p)$.

**b.** Betrachten Sie das unten abgebildete Flussnetzwerk. Die Kanten sind mit ihren Kapazitäten beschriftet. Berechnen Sie mit dem Ford Fulkerson Algorithmus den maximalen Fluss von der Quelle $s$ zur Senke $t$. Geben Sie dabei für jeden Schritt den augmentierenden Pfad in Form einer Knotenliste und den Fluss, den Sie über diesen Pfad schieben, an. Geben Sie zusätzlich den Wert des maximalen Flusses nach der Ausführung an. [3 Punkte]




## Lösung

- sat: 1
- sbt: 2
- sbct: 1
- sbcat: 1

Gesamtfluss: 5

**c.** In einem Online-Shop gibt es bunte Spielsteine aus Holz zu kaufen. Die Spielsteine werden in Packungen zu je $n$ Steinen verkauft und können bis zu $n$ unterschiedliche Farben haben. Die Farben der Spielsteine in jeder Packung sind unabhängig und gleichverteilt zufällig. Wie viele Packungen müssen Sie asymptotisch in Erwartung kaufen, um mindestens einen Stein jeder Farbe zu haben? Begründen Sie kurz. [2 Punkte]

## Lösung

Dadurch, dass die Steine unabhängig zufällig sind, können wir das ganze auch einfach als simples Cupon Collector Problem betrachten. Wie in der Übung gesehen, erwarten wir, dass $\mathcal{O}(n \log n)$ Steine betrachtet werden müssen, um von jeder Farbe einen zu haben. Dies entspricht $\mathcal{O}\left(\frac{n \log n}{n}\right) = \mathcal{O}(\log n)$ Packungen. Exakter: $n$-te Harmonische Zahl ($H_n$) Packungen.

**d.** Nennen Sie drei der online-Strategien, die in der Vorlesung für das Online-Paging Problem betrachtet wurden. Geben Sie zusätzlich zu jeder Strategie an, ob die Strategie für dieses Problem einen endlichen kompetitiven Faktor hat. [2 Punkte]

## Lösung

| Strategie | endlicher komp. Faktor? |
|---|---|
| LIFO | nein |
| FIFO | ja |
| LFU | nein |
| LRU | ja |
| FWF | ja |
| RMARK | ja |

Lösungsvorschlag

**Aufgabe 2.** Priority Queues [11 Punkte]

**a.** Ordnen Sie die folgenden Priority Queues ihren jeweiligen amortisierten Laufzeiten zu. Die Anzahl der Eingabe-Elemente sei $n$. Die Prioritäten seien Ganzzahlen aus $[0, C]$. [2 Punkte]

## Lösung

| Priority Queue | $T_{\text{insert}}$ | $T_{\text{deleteMin}}$ | $T_{\text{decreaseKey}}$ |
|---|---|---|---|
| A Sortierte verkettete Liste | $\Theta(n)$ | $\Theta(1)$ | $\Theta(n)$ |
| C Pairing Heap | $\Theta(1)$ | $\Theta(\log n)$ | $\mathcal{O}(\log n)$ |
| E Bucket Queue | $\Theta(1)$ | $\Theta(C)$ | $\Theta(1)$ |
| B Unsortierte verkettete Liste | $\Theta(1)$ | $\Theta(n)$ | $\Theta(1)$ |
| D Fibonacci Heap | $\Theta(1)$ | $\Theta(\log n)$ | $\Theta(1)$ |
| F Radix Heap | $\Theta(1+\log C)$ | $\Theta(1+\log C)$ | $\Theta(1)$ |

**b.** Gegeben sei ein Graph mit $n$ Knoten und $m$ Kanten. Die Kantengewichte seien ganzzahlig aus dem Intervall $[0, m^2]$ und es sei $m \in \Theta(\log n)$. Mit welcher der oben angegebenen Priority Queues hat der Dijkstra-Algorithmus die beste asymptotische Laufzeit? Geben Sie die Laufzeit mit dieser Priority Queue an. [1 Punkt]

## Lösung

| Priority Queue | $T_{\text{dijkstra}}$ | Eingesetzt |
|---|---|---|
| A Sortierte verkettete Liste | $(n+m)m$ | $n \log n$ |
| C Pairing Heap | $(n+m)\log n$ | $n \log n$ |
| E Bucket Queue | $m+nC$ | $n(\log n)^2$ |
| B Unsortierte verkettete Liste | $m+n^2$ | $n^2$ |
| D Fibonacci Heap | $m+n\log n$ | $n \log n$ |
| F Radix Heap | $m+n\log C$ | $n \log\log n$ |

Also ist ein Radix Heap in diesem Fall am besten.

**c.** Beschreiben Sie, wie es möglich ist, durch die Operationen `insert`, `deleteMin` und `decreaseKey` einen Fibonacci Heap zu erzeugen, der Größe $n = 2^k, k \in \mathbb{N}$ hat, und bei dem $n-1$ Knoten markiert sind.
*Hinweis*: Wurzeln können nicht markiert werden. [3 Punkte]

## Lösung

- `insert` 0,...,$2n$
- `deleteMin`
- Abwechselnd $n$-mal `decreaseKey` + `deleteMin` auf Handle 2, 4, 6, ..., $2n$

In einem Sweepline-Algorithmus werden aus den $n$ Eingabewerten Events erstellt. Die Liste der abzuarbeitenden Events werde in einer Priority Queue verwaltet. Der Algorithmus erzeugt während der Ausführung zusätzliche Events. Sie können davon ausgehen, dass zu keinem Zeitpunkt mehr als $k$ dieser zusätzlichen Events in der Priority Queue sind. Die Verarbeitung jedes Events (bis auf die Queue-Operationen) benötigt konstante Zeit. Insgesamt werden im Algorithmus $\mathcal{O}(n)$ Events verarbeitet und pro Event wird maximal eine konstante Anzahl Operationen auf der Priority Queue ausgeführt.

**d.** Als Priority Queue werde ein Fibonacci Heap verwendet. Geben Sie die Laufzeit des gesamten Sweepline-Algorithmus an und begründen Sie kurz. [2 Punkte]

## Lösung

In jedem Schritt des Algorithmus findet eine PQ-Operation statt, um das nächste Event zu finden. Dies benötigt Zeit $\mathcal{O}(\log(n+k))$. Das Event wird in konstanter Zeit verarbeitet. Möglicherweise wird auch ein neues Event in die PQ eingefügt, was Zeit $\mathcal{O}(1)$ benötigt. Insgesamt ergibt sich Zeit $\mathcal{O}(n\log(n+k))$.
Da pro Event maximal eine konstante Anzahl PQ-Operationen ausgeführt wird (laut Aufgabe), ist die Größe der Queue (unabhängig von $k$) in $\mathcal{O}(n)$. Dadurch lässt sich die gesamte Laufzeit auch durch $\mathcal{O}(n\log n)$ abschätzen.

**e.** Seien nun die $n$ Eingabe-Elemente schon nach ihrer Priorität sortiert gegeben. Beschreiben Sie eine alternative Priority Queue, die sich diesen Fakt zu Nutze macht. Mit der neuen Priority Queue soll die Laufzeit des Sweepline-Algorithmus nun $\mathcal{O}(n\log k)$ sein. Begründen Sie kurz die Laufzeit. [3 Punkte]

## Lösung

Verwalte 2 Datenstrukturen: Eine Priority Queue für die “dynamischen” Events und eine sortierte Liste für die Events, die aus der Eingabe entstanden sind. Bei einem *deleteMin* vergleiche das Minimum der PQ und das nächste Element in der Liste. Gib das Minimum der beiden zurück. Neue Events werden in die PQ eingefügt.
Pro Iteration findet eine konstante Anzahl PQ-Operationen statt, wobei die Queue maximal Größe $k$ hat. Durch Verwendung einer geeigneten PQ (z.B. Fibonacci Heaps) ergibt sich eine Laufzeit von $O(\log k)$ pro Operation. Des Weiteren findet ein Vergleich mit dem ersten Element der sortierten Liste statt, was Zeit $O(1)$ benötigt. Insgesamt passiert dies $n$ mal, wodurch sich die Laufzeit $O(n\log k)$ ergibt.

Lösungsvorschlag

EK

ZK

**Aufgabe 3.** Externe Algorithmen: Wissenschaftliche Papers [12 Punkte]

Gegeben seien:

- Eine Liste $P$ mit Papers und ihren jeweiligen Autoren (Paper X $\rightarrow$ {Autor $\alpha$, $\beta$, $\gamma$}, ...)
- Eine Liste $Z$ mit Zitationen (Paper X zitiert Paper Y, ...). Jede Zitation besteht aus genau einem zitierenden und einem zitierten Paper.

Die Paper-Titel sowie die Namen der Autoren seien eindeutig. Die Listen seien zu lang für den internen Speicher der Größe M. Deshalb liegen sie auf einer Festplatte, die in Blöcken der Größe $B < M$ zugegriffen werden kann. Der Einfachheit halber haben alle Listeneinträge der beiden Listen jeweils die gleiche, konstante Länge. Beide Listen-Längen lassen sich durch $n$ abschätzen. Beide Listen haben keine bestimmte Reihenfolge.

**a.** Geben Sie einen Algorithmus an, der alle Papers bestimmt, die ein gegebener Autor geschrieben hat. Die Ausgabe-Liste kann zu groß für den internen Speicher werden und soll deshalb im externen Speicher liegen. Sie dürfen dafür $\mathcal{O}(n/B)$ I/O-Operationen verwenden. Begründen Sie die Anzahl der I/O-Operationen kurz. [2 Punkte]

## Lösung

Wir verwenden 2 Puffer der Größe $B$ im internen Speicher, einen für die Eingabe und einen für die Ausgabe. Wir laden der Reihe nach alle Blöcke der Liste $P$ in den Eingabe-Puffer. Dann iterieren wir im internen Speicher über die Papers im Block und sehen nach, bei welchen davon der angefragte Autor dabei ist. Die Überprüfung ist in konstanter Zeit möglich, weil alle Listeneinträge eine konstante Länge haben. Diese Papers des Autors schreiben wir dann in den Ausgabe-Puffer. Sobald der Ausgabe-Puffer voll ist, schreiben wir ihn in den externen Speicher und leeren ihn. Ganz am Ende, nachdem wir durch die komplette Liste $P$ iteriert sind, schreiben wir den möglicherweise angefangenen Ausgabe-Puffer in den externen Speicher.
Die Laufzeit ergibt sich daraus, dass jeder Block der Eingabe maximal 1x gelesen werden muss ($\lceil n/B \rceil$ I/Os) und maximal wieder $\lceil n/B \rceil$ Blöcke geschrieben werden müssen.

**b.** Für eine Meta-Studie soll bestimmt werden, wie oft welche Wörter in Paper-Titeln vorkommen. Geben Sie einen Algorithmus an, der die Anzahl an Vorkommen jedes Wortes bestimmt. Ihr Algorithmus darf $\mathcal{O}\left(\frac{n}{B}\left(1+\log_{M/B}\left(\frac{n}{M}\right)\right)\right)$ I/O-Operationen benutzen. Die Anzahl verschiedener Wörter sei zu groß, um sie alle im internen Speicher zu halten. [2 Punkte]

## Lösung

Scanne einmal linear über die Liste $P$ und extrahiere eine Liste aller Wörter in Titeln. Verfahre dabei ähnlich wie in Teilaufgabe a, also verwende zwei Puffer. Sortiere diese Liste an Wörtern alphabetisch, z.B. mit external memory merge sort. Scanne dann linear über die Wörter. Verwende die Eigenschaft, dass nun alle Vorkommen eines Wortes direkt hintereinander kommen. Inkrementiere also so lange einen Zähler, wie das gleiche Wort gelesen wird. Wird ein neues Wort gelesen oder ist die Liste zu Ende, gib den aktuellen Zähler aus.

**c.** Sie haben im externen Speicher eine Liste $H$ der Häufigkeiten aller Wörter in Titeln von Papers gegeben. Die Liste habe Größe $\ell$ und sei zu groß für den internen Speicher. Die Häufigkeiten seien paarweise verschieden. Geben Sie einen Algorithmus an, der die $k$ häufigsten

Wörter bestimmt. Ihr Algorithmus darf $\mathcal{O}\left(\frac{\ell}{B}\right)$ I/O-Operationen und $\mathcal{O}(\ell \log k)$ interne Arbeit benutzen. Die Anzahl $k$ sei klein genug, dass problemlos $k$ Wörter in den internen Speicher passen. [3 Punkte]

## Lösung

Erstelle eine minimum Priority Queue im internen Speicher. Füge zunächst die ersten $k$ Einträge in die Priority Queue ein und scanne dann linear über den Rest. Falls die betrachtete Häufigkeit kleiner als das Minimum der Queue ist, verwerfe das zugehörige Wort. Falls sie größer als das Minimum ist, führe *deleteMin* aus und füge stattdessen diese ein. Am Ende stehen in der Queue die $k$ häufigsten Wörter.

Um zu bestimmen, ob die Papers eines Autors sowohl Masse als auch Klasse haben, betrachten wir den h-Index eines Autors. Der h-Index $h$ eines Autors ist das maximale $h$, sodass $h$ seiner Papers *jeweils* mindestens $h$-mal zitiert wurden.

**d.** Ein Autor hat 4 Papers geschrieben, die 1, 2, 8, und 300 mal zitiert wurden. Geben Sie den h-Index des Autors an. [1 Punkt]

## Lösung

2

**e.** Geben Sie einen Algorithmus an, der aus den extern gespeicherten Listen $P$ und $Z$ für einen gegebenen Autor dessen h-Index bestimmen kann. Ihr Algorithmus darf $\mathcal{O}\left(\frac{n}{B}\left(1+\log_{M/B}\left(\frac{n}{M}\right)\right)\right)$ I/O-Operationen benutzen. Begründen Sie die Anzahl der I/O-Operationen kurz. [4 Punkte]

## Lösung

Hier gibt es mehrere ähnliche Lösungen. Eine Möglichkeit:

- Bestimme für jedes Paper, wie oft es zitiert wurde. Dazu sortiere die Zitations-Liste lexikographisch aufsteigend nach dem zitierten Paper. Dann scanne darüber und summiere die Anzahl der Zitationen jedes Papers auf (ähnlich wie in Teilaufgabe b, wo die Vorkommen von Wörtern gezählt wurden).
- Bestimme alle Papers des Autors durch einen Scan (wie in Teilaufgabe a).
- Sortiere die Papers des Autors, sowie die Liste der Zitations-Häufigkeiten Papers lexikographisch aufsteigend. Scanne gemeinsam die Liste der Zitations-Häufigkeiten und der Papers des Autors, um nur Zitationen der Papers des entsprechenden Autors herauszufinden.
- Sortiere diese Liste nach Anzahl der Zitate, das Paper mit den meisten Zitaten kommt zuerst.
- Scanne über diese Liste, solange an Array-Index $i$ noch mehr als $i$ Zitationen stehen. Ist das nicht mehr der Fall, ist der $h$-Index durch $(i-1)$ gegeben.

Als Bausteine scannen wir mehrere Listen und sortieren mehrere Listen. Da die Länge aller Listen durch $n$ abgeschätzt werden kann, erhalten wir so viele I/O-Operationen wie der external memory merge sort, was die geforderte Anzahl ist.

Lösungsvorschlag

**Aufgabe 4.** Parametrisierte Algorithmen: Konjunktive Normalform [7 Punkte]

Es sei eine aussagenlogische Formel in konjunktiver Normalform (KNF) über einem Universum von (nur positiven) Literalen $S$ gegeben. Die Formel ist definiert durch eine Menge von Klauseln $C = \{c \subseteq S : 0 < |c| \leq 5\}$ der Größe $|C| = n$. Jede Klausel $c \in C$ ist also eine nicht-leere Menge von bis zu fünf Literalen. Eine Klausel stellt eine Disjunktion ihrer Literale dar und die Formel besteht aus der Konjunktion aller Klauseln. Die KNF enthält in dieser Aufgabe keine Negation.

Beispiel: Die Klauselmenge

$$C = \{\{l_1, l_2\}, \{l_2, l_3, l_4, l_5, l_6\}, \{l_1, l_5, l_6\}\}$$

entspricht der aussagenlogischen Formel

$$F = (l_1 \vee l_2) \wedge (l_2 \vee l_3 \vee l_4 \vee l_5 \vee l_6) \wedge (l_1 \vee l_5 \vee l_6)$$

Eine Lösung $L \subseteq S$ der KNF ist eine Menge von Literalen, sodass jede Klausel in $C$ erfüllt ist. Eine Klausel $c \in C$ gilt dabei als erfüllt, wenn mindestens eines ihrer Literale in $L$ enthalten ist, also $c \cap L \neq \emptyset$. Es soll entschieden werden, ob eine Lösung für $C$ existiert, welche genau $k$ Literale enthält.

**a.** Gegeben sei folgende Klauselmenge:

$$C = \{\{l_1, l_2, l_3, l_4\}, \{l_2, l_5, l_6\}, \{l_6\}, \{l_3, l_5\}\}.$$

Existiert eine Lösung für diese Instanz des Entscheidungsproblems mit $k = 2$? Falls ja, geben Sie eine Lösung der Größe 2 an, falls nein, begründen Sie. [2 Punkte]

## Lösung

Lösung der Größe $k = 2$: $\{l_3, l_6\}$

**b.** Geben Sie einen FPT (fixed parameter tractable) Algorithmus an, der das Entscheidungsproblem für beliebiges $k$ löst. Geben Sie auch die asymptotische Laufzeit Ihres Algorithmus in Abhängigkeit von $k$ und $n$ an und begründen Sie diese. Gehen Sie darauf ein, wieso Ihr Algorithmus FPT ist. Zur Erinnerung: Jede Klausel $c \in C$ ist eine nicht-leere Menge von bis zu fünf Literalen. [5 Punkte]

## Lösung

Der folgende Algorithmus löst das Entscheidungsproblem:

**Algorithmus 1** `kLiteralSolution(`$C = \{c_1,\ldots,c_n\}$`: Menge von nicht erfüllten Klauseln, `$k$`: Anzahl zu setzender Literale)`

```
if C = ∅ then
    return True
else if k = 0 then
    return False
for all Literale l ∈ c_1 do                    ▷ Maximaler Verzweigungsgrad 5.
    C' ← ∅
    for all c ∈ C do                           ▷ Klauseln aus C ohne Literal l.
        if c ∩ {l} = ∅ then
            C' ← C' ∪ {c}
    success ← kLiteralSolution(C', k − 1)
    if success then
        return True
return False
```

Laufzeit: Der Algorithmus baut einen Suchbaum der Tiefe $k$ auf. Der Verzweigungsfaktor ist immer kleiner gleich der maximalen Anzahl Literale in einer Klausel. Da eine Klausel aus maximal 5 Literalen besteht, ist der Verzweigungsfaktor kleiner oder gleich 5. Die neue Menge von Klauseln $C'$ kann in Zeit $O(n)$ Zeit konstruiert werden, da die Größe von jeder der $n$ Klauseln durch eine Konstante (hier 5) beschränkt ist. Damit ergibt sich eine asympotische Laufzeit von $O(5^k \cdot n)$.
FPT: Die Laufzeit des Algorithmus hat die Form $f(k) \cdot \text{poly}(n)$, wobei $f$ eine berechenbare Funktion ist.

Lösungsvorschlag

EK

ZK

**Aufgabe 5.** Stringology: Suffix-Array- und LCP-Array-Konstruktion [10 Punkte]

**a.** Gegeben sei der Text $T =$ `simsalabim$`. Geben Sie Suffix-Array und LCP-Array an.

*Zwei Kopien. Wenn Sie mehr als ein Array beschriften, machen Sie deutlich, welche Kopie korrigiert werden soll. Andernfalls wird diese Teilaufgabe mit 0 Punkten bewertet.* [2 Punkte]

## Lösung

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $T[i]$ | s | i | m | s | a | l | a | b | i | m | $ |

| $SA[i]$ | 11 | 7 | 5 | 8 | 9 | 2 | 6 | 10 | 3 | 4 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $LCP[i]$ | 0 | 0 | 1 | 0 | 0 | 2 | 0 | 0 | 1 | 0 | 1 |

**b.** Geben Sie einen Text über dem kleinst möglichen Alphabet an, für den das Suffix-Array die folgende Form hat.

1. SA $= [9,8,7,6,5,4,3,2,1]$
2. SA $= [9,1,2,3,4,5,6,7,8]$

Achten Sie darauf, dass das letzte Zeichen des Textes ein Sentinel (`$`) ist und dass das Suffix-Array in dieser Aufgabe mit 1 indiziert ist. [2 Punkte]

## Lösung

1. `aaaaaaaa$`
2. `aaaaaaab$`

**c.** Führen Sie für den Text $T =$ `tolleknolle$` den Prefix Doubling Algorithmus aus. Geben Sie in jeder Iteration die vorläufigen Ränge der Suffixe an. [3 Punkte]

## Lösung

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $T[i]$ | t | o | l | l | e | k | n | o | l | l | e | $ |

Iteration

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 6 | 5 | 3 | 3 | 1 | 2 | 4 | 5 | 3 | 3 | 1 | 0 |
| 2 | 8 | 7 | 5 | 4 | 2 | 3 | 6 | 7 | 5 | 4 | 1 | 0 |
| 3 | 10 | 9 | 7 | 5 | 2 | 3 | 8 | 9 | 6 | 4 | 1 | 0 |
| 4 | 11 | 10 | 7 | 5 | 2 | 3 | 8 | 9 | 6 | 4 | 1 | 0 |

**d.** Sei $T$ ein Text der Länge $n$ und $SA$ und $LCP$ das Suffix- bzw. LCP-Array des Textes. Der größte Wert im LCP-Array wird als $maxLCP = \max\{LCP[i] : 1 \leq i \leq n\}$ bezeichnet. Sei darüber hinaus $SA_h$ ein Suffix-Array von $T$, in dem die Suffixe nach der $h$-Ordnung sortiert sind. $LCP_h$ ist dann das zu $SA_h$ gehörige LCP-Array. Beachten Sie, dass das LCP-Array über den gesamten Suffixen berechnet wird und nicht nur über den ersten $h$ Zeichen. Zeigen Sie, dass für $h < maxLCP$ gilt

$$\sum_{i=1}^{n} \mathrm{LCP}_h[i] \leq \sum_{i=1}^{n} \mathrm{LCP}[i].$$

[3 Punkte]

## Lösung

In einem partiellen Suffix-Array sind noch nicht alle Suffixe an ihrer korrekten Position. Wenn in einem Suffix-Array zwei korrekte Suffixe vertauscht werden, dann wird der damit korrespondierende LCP-Wert im LCP-Array eventuell kleiner. Wichtig ist, dass der Wert niemals größer wird, denn einen größeren Wert, als bei der korrekten lexikographischen Sortierung, gibt es nicht. Daher ist die Summe aller LCP-Werte niemals größer als beim komplett korrekt sortierten Suffix-Array. Da die partiellen Suffix-Arrays immer eine Teilweise falsche Sortierung haben, ist die Summe der LCP-Werte für ein partielles Suffix-Array entsprechend niemals größer.

Lösungsvorschlag

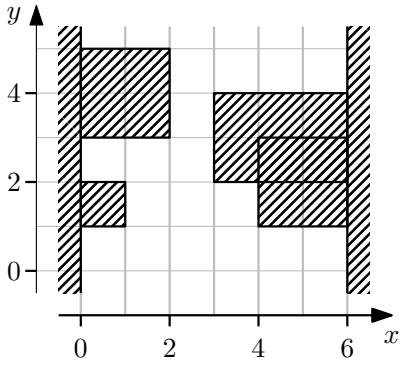
**Aufgabe 6.** Geometrische Algorithmen: "Quadro-Bot" [11 Punkte]

Es sei ein Gang in der $(x,y)$-Ebene gegeben. Der Gang sei definiert als ein Band zwischen $x = 0$ und $x = x_{\max} \in \mathbb{N}$ mit unendlicher Ausdehung in $y$-Richtung.

An den Wänden des Ganges befinden sich eine Menge linker Hindernisse $L \subseteq \mathbb{N}^3$ und eine Menge rechter Hindernisse $R \subseteq \mathbb{N}^3$. Jedes Hindernis ist ein achsenparalleles Rechteck. Ein linkes Hindernis $(y_0, y_1, w) \in L$ liegt zwischen $y = y_0$ und $y = y_1$ und ragt mit einer Breite von $w$ an der linken Wand in den Gang hinein. Ein rechtes Hindernis $(y_0, y_1, w) \in R$ liegt zwischen $y = y_0$ und $y = y_1$ und ragt mit einer Breite von $w$ an der rechten Wand in den Gang hinein. Hindernisse dürfen sich überschneiden.

Beispiel: In der Abbildung rechts ist ein Gang für $x_{\max} = 6$ und unten gegebene $L$ und $R$ dargestellt. Die Wände des Ganges und die Hindernisse sind schraffiert.

$L = \{(1,2,1),\ (3,5,2)\}$
$R = \{(1,3,2),\ (2,4,3)\}$




**a.** Sei $x_{\max}$ beliebig, aber fest. Geben Sie zunächst einen Algorithmus an, der feststellt, ob der Gang vollständig blockiert ist. Der Gang gelte dabei als vollständig blockiert, wenn sich an einer beliebigen Stelle des Ganges ein linkes und ein rechtes Hindernis berühren oder überlappen. Die Laufzeit Ihres Algorithmus soll in $\mathcal{O}(n \log n)$ liegen, wobei $n = |L| + |R|$. Begründen Sie, warum Ihr Algorithmus diese Laufzeit erreicht. [4 Punkte]

## Lösung

Wir verwenden einen Sweepline-Algorithmus.
Wir konstruieren zunächst eine Liste $E$ der Ereignisse. Füge für jedes Hindernis $l = (y_0, y_1, w) \in L$ ein Startereignis $(y_0, \text{“b”}, \text{“l”}, w)$ und ein Endereignis $(y_1, \text{“e”}, \text{“l”}, w)$ zu $E$ hinzu. Füge analog für jedes Hindernis $r = (y_0, y_1, w) \in R$ ein Startereignis $(y_0, \text{“b”}, \text{“r”}, w)$ und ein Endereigns $(y_1, \text{“e”}, \text{“r”}, w)$ zu $E$ hinzu. Sortiere $E$ nach dem ersten Tupel-Element, der $y$-Koordinate der Ereignisse. Bei Gleichheit zweier Ereignisse in der $y$-Koordinate, kommen Startereignisse (“b”) vor Endereignissen (“e”) und oBdA linke Ereignisse (“l”) vor rechten Ereignissen (“r”).

Zu jedem Zeitpunkt verwalten wir als Sweepline-Status jene Hindernisse, die von der Sweepline geschnitten werden, in einer adressierbaren Max-PQ $Q_l$ für linke Hindernisse bzw einer addressierbaren Max-PQ $Q_r$ für rechte Hindernisse. Füge initial 0 zu beiden PQs hinzu. Iteriere dann über die sortierten Elemente in $E$ und gehe wie folgt vor:

- Sei $e = (y, t, d, w)$ das nächste Ereignis.
- Falls $t =$ “b”:
  - Falls $d =$ “l”, füge $w$ in $Q_l$ ein.
  - Falls $d =$ “r”, füge $w$ in $Q_r$ ein.
  - Falls $Q_r.\texttt{max}() + Q_l.\texttt{max}() \geq x_{\text{max}}$, ist der Gang bei $y$ blockiert. Gib falsch zurück.
- Falls $t =$ “e”:
  - Falls $d =$ “l”, entferne $w$ aus $Q_l$.
  - Falls $d =$ “r”, entferne $w$ aus $Q_r$.

Konnten alle Hindernisse abgearbeitet werden, gib wahr zurück.

**Laufzeit:** Zunächst werden $2n$ Ereignisse vergleichsbasiert in $\mathcal{O}(n \log n)$ sortiert. Die Sweepline verarbeitet $2n$ Ereignisse und braucht pro Ereignis $\mathcal{O}(\log n)$ Zeit, um ein Element in die entsprechende PQ hinzuzufügen oder ein Element aus der PQ zu entfernen. Damit liegt die Gesamtlaufzeit bei $\mathcal{O}(n \log n)$.

**Alternativer Ansatz:** Verwendung des Plane-Sweep Algorithmus für orthogonalen Streckenschnitt aus VL. Man wendet Algorithmus aus VL zwei Mal an:

- erster Durchlauf: vertikale Geraden = rechte Seiten der linken Hindernisse und linke Wand; horizontale Geraden = untere, obere Seiten der rechten Hindernisse
- zweiter Durchlauf: vertikale Geraden = linke Seiten der rechten Hindernisse und rechte Wand; horizontale Geraden = untere, obere Seiten der linken Hindernisse

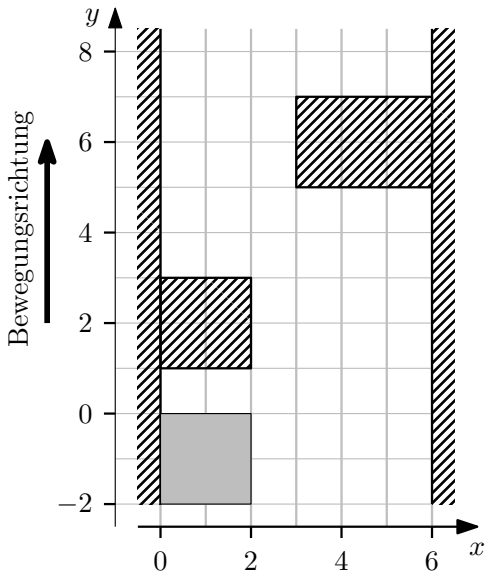
Der Algorithmus aus der Vorlesung deckt die Sonderfälle für gleiche $y$-Koordinaten ab. Die Laufzeit des Algorithmus aus der Vorlesung liegt in $\mathcal{O}(n \log n + m)$, wobei $m$ die Anzahl der ausgegebenen Schnitte ist. Da hier beim ersten Schnitt abgebrochen werden kann, wird die geforderte Laufzeit also eingehalten.

## Lösung

Der Roboter “Quadro-Bot” soll einen Pfad durch den Gang mit Hindernissen finden. Quadro-Bot sei ein achsenparelleles Quadrat mit Seitenlänge $k \in \mathbb{N}$. Der Roboter kann sich in $x$- und $y$-Richtung bewegen, aber kann sich nicht drehen.
Quadro-Bot startet unterhalb aller Hindernisse und soll sich in positiver $y$-Richtung an diesen vorbei bewegen. Der Roboter darf während der Bewegung die Wände oder Hindernisse berühren, aber darf sich nie mit ihnen überschneiden.

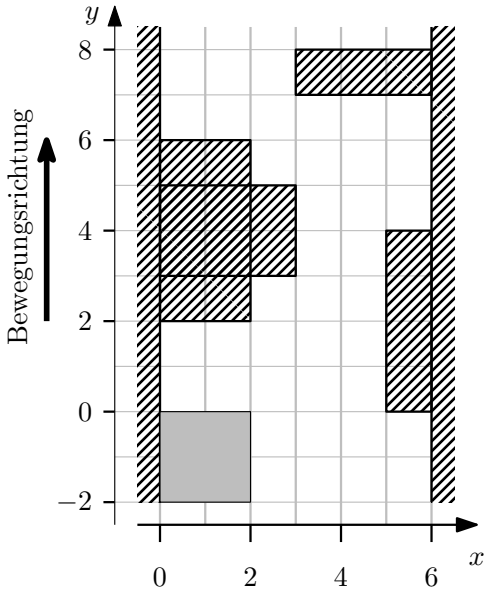
**b.** In den beiden unten dargestellten Beispielen gelte $k = 2$ und $x_{\max} = 6$. Die Startposition von Quadro-Bot ist grau gezeichnet und Hindernisse sind schraffiert. Geben Sie für beide Beispiele an, ob Quadro-Bot sich an allen Hindernissen vorbei bewegen kann und $y = 8$ erreichen kann.

Beispiel 1:
$L = \{(1,3,2)\}$
$R = \{(5,7,3)\}$




Kann $y = 8$ erreichen? ☐ ja ☐ nein

Beispiel 2:
$L = \{(2,6,2),\ (3,5,3)\}$
$R = \{(0,4,1),\ (7,8,3)\}$




Kann $y = 8$ erreichen? ☐ ja ☐ nein

[2 Punkte]

## Lösung

| Beispiel 1: ja | Beispiel 2: nein |
|---|---|

**c.** Sei $k$ nun beliebig, aber fest. Geben Sie einen Algorithmus an, welcher für beliebige Mengen von Hindernissen $L$ und $R$ bestimmt, ob Quadro-Bot sich an allen Hindernissen vorbei bewegen kann. Die Laufzeit Ihres Algorithmus soll in $\mathcal{O}(n \log n)$ liegen, wobei $n = |L| + |R|$. Begründen Sie, warum Ihr Algorithmus diese Laufzeit erreicht. [3 Punkte]

## Lösung

Modifikation des Algorithmus, der Teilaufgabe **a.** löst:

- Verschiebe $y_1$-Wert jedes Hindernisses um $+k$ beim Erzeugen der End-Events
- Bei Gleichheit zweier Ereignisse in der $y$-Koordinate kommen Endereignisse jetzt vor Startereignissen (Berührung erlaubt)
- Nach jedem Startevent: Falls $Q_l.\texttt{max}() + Q_r.\texttt{max}() + k > x_{\max}$, kann Quadro-Bot nicht passieren.

Die Laufzeit des Algorithmus ändert sich nicht, da für jedes Event nur maximal konstante zusätzliche Arbeit verrichtet wird.

**Modifikation des alternativen Ansatzes:**

Man kann weiterhin zwei Durchläufe des Planesweep-Algorithmus aus der Vorlesung verwenden. Dazu nimmt man die folgenden Änderungen vor:

- Verlängerung der hor. Geraden um $k$ von der Wand weg
- Verlängerung der vertikalen Geraden um $k$ in pos. $y$-Richtung
- Änderung des Tie-Breaking: Erst vertikale End-Events, dann horizontale Events, dann vertikale Start-Events (da Berührungen erlaubt)

Analog zu **a.**, hat der Algorithmus Laufzeit $O(n \log n)$.

## Lösung

**d.** Es seien $x_{\max}$ sowie $L, R$ beliebig, aber fest. Es sei außerdem ein Algorithmus bekannt, der für beliebiges $k$ in Laufzeit $T$ entscheidet, ob Quadro-Bot sich an allen Hindernissen vorbei bewegen kann.

Geben Sie einen Algorithmus an, der das größte $k \in \mathbb{N}$ bestimmt, sodass Quadro-Bot sich an allen Hindernissen vorbei bewegen kann. Die Laufzeit Ihres Algorithmus soll in $\mathcal{O}(T \log x_{\max})$ liegen. Begründen Sie, warum Ihr Algorithmus diese Laufzeit erreicht. [2 Punkte]

## Lösung

Wir verwenden binäre Suche:

- Setze initial $k_{\min} \leftarrow 0$, $k_{\max} \leftarrow x_{\max} + 1$.
- Solange $k_{\min} + 1 < k_{\max}$:
  - Setze $k \leftarrow \lceil (k_{\min} + k_{\max})/2 \rceil$
  - Falls Quadro-Bot für $k$ an allen Hindernissen vorbei kommt, setze $k_{\min} \leftarrow k$.
  - Sonst, setze $k_{\max} \leftarrow k$.
- Gebe $k_{\min}$ zurück.

Laufzeit: Binäre Suche in $\mathcal{O}(\log x_{\max})$ Iterationen, dabei jeweils $\mathcal{O}(T)$ für die Ausführung des geg. Algorithmus pro Iteration.