Algorithmen & Datenstrukturen
Midterm Test 2

Martin Avanzini <martin.avanzini@uibk.ac.at>
Thomas Bauereiß <thomas.bauereiss@uibk.ac.at>
Herbert Jordan <herbert@dps.uibk.ac.at>
René Thiemann <rene.thiemann@uibk.ac.at>

14. Juni

Name	
Matrikelnummer	

Aufgabe	**mögliche Punkte**	**erreichte Punkte**
1.1	10	
1.2	5	
2.1	7	
2.2	5	
2.3	8	
3.1	7	
3.2	3	
Gesamt	**45**	

Aufgabe 1) Mengen (15 Punkte) Ein Algorithmus benötigt für seine Abarbeitung eine effiziente Datenstruktur `Set` zur Verwaltung einer Menge von natürlichen Zahlen (`int`). Folgende Operationen sollen mit der gegebenen Laufzeitkomplexität auf einer Menge A ausgeführt werden können:

- Einfügen - im Durchschnitt $\mathcal{O}(1)$
- Löschen - im Durchschnitt $\mathcal{O}(log(|A|))$
- Member-Ship Test (`contains(int x)`) - im Durchschnitt $\mathcal{O}(1)$
- Iteration über alle Elemente - im Durchschnitt $\mathcal{O}(|A|)$
- Durchschnitt zweier Mengen A und B (`intersect(Set A, Set B)`) - im Durchschnitt $\mathcal{O}(min(|A|, |B|))$

1. Wählen Sie eine geeignete Datenstruktur für die Realisierung der Menge. Beschreiben Sie alle notwendige Details. Begründen Sie, weshalb Ihre Wahl den Anforderungen (Laufzeiten) entspricht.
2. Implementieren Sie in Java-ähnlicher Syntax die Funktion `intersect(Set A, Set B)` zur Berechnung des Durchschnitts der Mengen `A` und `B`. Notwendige Operationen wie `contains(int x)`, `add(int x)`, Iteratoren, usw. können Sie als gegeben voraussetzen. Begründen Sie, dass Ihre Implementierung eine Laufzeit von $\mathcal{O}(min(|A|, |B|))$ hat.

Aufgabe 1) Lösung

1. Eine entsprechende Datenstruktur: Eine Hashtable bei der der Schlüssel gleichzeitig dem zu speichernden Wert entspricht. Die einzelnen Element innerhalb der Tabelle sind doppelt verkettet. Zusätzlich zur Tabelle wird ein Verweis auf das zuletzt eingefügte Element verwaltet. Um effizient die Anzahl der Element in der Menge zu erhalten, wird diese noch extra in einer Variable `size` mitgeschrieben.

   Laufzeiten:

   (a) Einfügen - Einfügen in die Hashtable ($\mathcal{O}(1)$), einbinden in die Verknüpfte Liste der Elemente unter Zuhilfenahme des zuletzt eingefügten Elementes ($\mathcal{O}(1)$), updaten des zuletzt eingefügten Elements ($\mathcal{O}(1)$) sowie das Inkrementieren von `size` ($\mathcal{O}(1)$)

   (b) Löschen - Löschen aus der Hashtable $\mathcal{O}(1)$, Korrektur der Zeiger durch verknüpfen des Vorgängers und Nachfolgers $\mathcal{O}(1)$, überprüfen ob letztes eingefügte Element gelöschtes Element ist und gegebenenfalls Vorgänger ermitteln $\mathcal{O}(1)$ sowie dekrementieren von `size` ($\mathcal{O}(1)$)- insgesamt also $\mathcal{O}(1)$, wodurch der Aufwand auch in $\mathcal{O}(log(n))$ liegt.

   (c) Member-Ship Test - Lookup in Hashtable: $\mathcal{O}(1)$

   (d) Iteration über alle Elemente - unter Zuhilfenahme der Verkettung, rückwärts ausgehend vom letzten eingefügten Element - $\mathcal{O}(|A|)$

   (e) Durchschnitt - siehe Teil 2

2. Eine Mögliche Implementierung:

```
Set intersect(Set A, Set B) {
    // make sure A is smaller set
    if (A.size > B.size) {
        return intersect(B,A);
    }

    // compute intersection
    Set res = new Set();
    for(int cur : A) {
        if (B.contains(cur)) {
            res.add(cur);
        }
    }
    return res;
}
```

   Die Bedingung zu Beginn hat $\mathcal{O}(1)$. Sollte die Bedingung nicht erfüllt sein, hat die Implementierung eine Komplexität von $\mathcal{O}(|A|)$, da durch die Elemente von A iteriert wird ($\mathcal{O}(n)$) und der Membership Test sowie das Hinzufügen zum Resultat $\mathcal{O}(1)$ hat. Da in diesem Fall $|A| = min(|A|, |B|)$, liegt der Algorithmus in der geforderten Klasse. Sollte die erste Bedingung nicht erfüllt sein, hat die Ausführung der Methode den Aufwand des rekursiven Aufrufs. Innerhalb des Aufrufs gilt $|B| < |A|$, wodurch die Bedingung zu Beginn unerfüllt bleibt. Die Schleife hat wie im ersten Fall gezeigt einen Aufwand von $\mathcal{O}(n)$ wobei dieses mal $n = |B|$. Da $|B| = min(|A|, |B|)$ ergibt sich eine Komplexität von $\mathcal{O}(min(|A|, |B|))$.

Aufgabe 2) Dynamisches Programmieren (20 Punkte) Sei a ein Array über Integer-Werten der Länge $n > 0$. Eine *zusammenhängende Teilsequenz von a mit maximaler Summe* ist gegeben durch ein Paar (i, j) von Zahlen $0 \leqslant i \leqslant j < n$ sodass die Summe $\Sigma_{k=i}^{j} a[k]$ maximal ist. (Insbesondere gilt $\Sigma_{k=i'}^{j'} a[k] \leqslant \Sigma_{k=i}^{j} a[k]$ für jedes weitere Paar (i', j').)

Solch ein Paar lässt sich durch dynamisches Programmieren errechnen indem man folgende Rekursionsgleichung betrachtet:

$$\begin{aligned} U(0) &= a[0] \\ U(k+1) &= \max\{U(k) + a[k+1], a[k+1]\} \end{aligned}$$

Der Wert $U(k)$ beschreibt die maximale Summe einer zusammenhängenden Teilsequenz endend mit Index k. Eine Lösung (i, j) kann wie folgt berechnet werden:

- Der Index j wird so gewählt, dass $U(j)$ maximal ist (im Bereich $0 \leqslant j < n$).
- Zur Bestimmung des Anfangsindex i wird eine Abbildung $L\colon \{0, \ldots, n-1\} \to \mathsf{Int}$ errechnet sodass die Summe der Einträge von $a[L(j)], \ldots, a[j]$ maximal ist. Es gilt also $i = L(j)$. Beachten Sie, dass sich L anhand der Berechnung von U erstellen lässt.

1. Implementieren Sie die Berechung von U mittels einer Methode `int[] u(int a[])`.
2. Sei `a = {-1,2,3,-2}`. Bestimmen Sie die Werte $U(k)$ sowie $L(k)$ für $0 \leqslant k < 4$. Geben Sie die Lösung (i, j) an.
3. Erstellen Sie die Rekursionsgleichung für L anhand der Werte $U(k)$ und $a[k]$ $(0 \leqslant k < n)$.

Aufgabe 2) Lösung

1.

```
public class MaxInterval {
    public int[] u(int a[]) {
    int n = a.length;
    assert(n >= 1);

    int[] u = new int[n];
    u[0] = a[0];
    for(int k = 1; k < n; k++) {
        int v = u[k-1] + a[k];
        if (v > a[k]) { u[k] = v; } else { u[k] = a[k];}
    }
    return u;
    }
}
```

2. Die Werte für U und L sind in folgender Tabelle angeführt:

k	0	1	2	3
$U(k)$	-1	2	5	3
$L(k)$	0	1	1	1

   Die Lösung für Eingabearray `a` ist demzufolge $(1, 2)$.

3. Die Rekursionsformel sieht wie folgt aus:

$$L(0) = 0$$
$$L(k+1) = \begin{cases} L(k) & \text{wenn } U(k+1) = U(k) + a[k+1] \\ k+1 & \text{wenn } U(k+1) = a[k+1]. \end{cases}$$

Aufgabe 3) Topologische Sortierung (10 Punkte) Gegeben sei ein Graph $G = (V, E)$ mit

$$
\begin{aligned}
V = \{&a, b, c, d, e, f, g, h\} \\
E = \{&(a, g), \\
&(c, b), (c, d), (c, e), \\
&(d, b), \\
&(e, b), \\
&(f, a), (f, c), (f, h), \\
&(g, c), \\
&(h, g)\}
\end{aligned}
$$

1. Bestimmen Sie eine topologische Sortierung des Graphen G, indem Sie den in der Vorlesung vorgestellten effizienten Algorithmus anwenden. Geben Sie die Zwischenergebnisse an (z.B. in Form einer Tabelle) sowie die resultierenden Ordnungsziffern $ord(v)$ für alle Knoten $v \in V$.

2. Zeichnen Sie den Graphen G mit Hilfe der Ergebnisse der vorherigen Teilaufgabe. Platzieren Sie dabei die Knoten in der Reihenfolge der topologischen Sortierung entlang einer Geraden, und achten Sie darauf, dass sich die Kanten nicht überschneiden.

Aufgabe 3) Lösung

1. Die folgende Tabelle gibt die Zwischenschritte des Algorithmus von Folie 313 für den Graphen G an. Die Spalte `i` gibt die aktuelle Iteration der Schleife an, die Spalten $a - h$ die Werte des Arrays `indeg`, und die Spalte `indegZero` die Warteschlange mit zu bearbeitenden Knoten.

`i`	a	b	c	d	e	f	g	h	`indegZero`
0	1	3	2	1	1	0	2	1	f
1	0	3	1	1	1	0	2	0	a h
2	0	3	1	1	1	0	1	0	h
3	0	3	1	1	1	0	0	0	g
4	0	3	0	1	1	0	0	0	c
5	0	2	0	0	0	0	0	0	d e
6	0	1	0	0	0	0	0	0	e
7	0	0	0	0	0	0	0	0	b
8	0	0	0	0	0	0	0	0	

   Es ergeben sich folgende Ordnungsziffern:

v	a	b	c	d	e	f	g	h
$ord(v)$	2	8	5	6	7	1	4	3

2. Mit den Ergebnissen aus der vorhergehenden Teilaufgabe kann der Graph unter Berücksichtigung der topologischen Sortierung gezeichnet werden: