Probeklausur

Algorithmen und Datenstrukturen

Universität:	Technische Universität Berlin
Kurs/Modul:	Algorithmen und Datenstrukturen
Bearbeitungszeit:	120 Minuten
Erstellungsdatum:	September 6, 2025

Zielorientierte Lerninhalte, kostenlos!
Entdecke zugeschnittene Materialien für deine Kurse:

https://study.AllWeCanLearn.com

Algorithmen und Datenstrukturen

Bearbeitungszeit: 120 Minuten.

Aufgabe 1.

(a) Bestimmen Sie die Laufzeitkomplexität des folgenden Codes in der O-Notation:

```
for i = 1 bis n
  for j = 1 bis i
    s := s + a[i] ·b[j]
```

(b) Beschreiben Sie die Speicherkomplexität eines Arrays C der Länge n, wenn C[i] = A[i] + B[i] gilt. Gehen Sie dabei von worst-case aus.

(c) Erklären Sie kurz die amortisierte Laufzeit von Einfügen am Ende eines dynamischen Arrays, das bei Bedarf verdoppelt wird.

(d) Nennen Sie zwei Vorteile von Hash-Tabellen gegenüber Arrays bei Schlüsselzugriffen und geben Sie eine einfache Hash-Funktion an.

Aufgabe 2.

(a) Gegeben sei der ungerichtete Graph $G = (V, E)$ mit $V = \{1, 2, 3, 4, 5\}$ und $E = \{(1, 2), (1, 3), (2, 3), (2, 4$
Bestimmen Sie:

- die Anzahl der Kanten,
- den Grad jedes Knotens.

(b) Beschreiben Sie den BFS-Algorithmus. Geben Sie eine mögliche BFS-Reihenfolge der Knoten aus Startknoten 1 an, falls die Nachbarschaftslisten in der Reihenfolge der Ziffern sortiert sind.

(c) Betrachten Sie den gewichteten Graphen

$G = \big(V, E, w\big), \quad V = \{s, u, v, t\},\ E = \{(s, u), (s, v), (u, w), (v, w), (w, t)\},\ w(s, u) = 10,\ w(s, v) = 5,\ w(u,$

Bestimmen Sie die kürzesten Pfade von s nach t mit Dijkstra. Geben Sie die Distanz und den Pfad an.

(d) Welche Graph-Darstellung (Adjazenzliste oder Adjazenzmatrix) eignet sich besser für sehr dichte Graphen und warum?

Aufgabe 3.

(a) Betrachten Sie das Flussnetzwerk mit Knoten $\{s, a, b, t\}$ und Kapazitäten:

$$s \to a : 16, \quad s \to b : 13, \quad a \to b : 12, \quad b \to t : 20, \quad a \to t : 0.$$

Bestimmen Sie den Max-Flow von s nach t und nennen Sie eine zugehörige Min-Cut.

(b) Erklären Sie in wenigen Sätzen das Max-Flow-Min-Cut-Theorem.

(c) Beschreiben Sie kurz, wie das Max-Flow-Konzept auf Scheduling-Probleme übertragbar ist.

Aufgabe 4.

(a) Geben Sie die Grundoperationen einer Binär-Heap-basierenden Prioritätswarteschlange an (Insert, Extract-Min) und deren Worst-Case-Laufzeit.

(b) Erklären Sie kurz den Unterschied zwischen Heapsort und der klassischen Sortiermethode Mergesort hinsichtlich Stabilität und Speicherbedarf.

(c) Gegeben sei ein Array $a[1..n]$. Beschreiben Sie einen Algorithmus, der in $O(n \log n)$ die Anzahl der Inversionspaare $i < j$ mit $a[i] > a[j]$ bestimmt. Geben Sie die zentrale Idee und die grobe Vorgehensweise an.

Lösungen

Bearbeitungszeit: 120 Minuten.

(a) Lösung: Die innere Schleife läuft für jedes i genau i Mal. Die Gesamtanzahl der Iterationen ist

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2).$$

Damit ist die Laufzeit $\Theta(n^2)$ (O-Notation: $O(n^2)$).

(b) Lösung: Der Speicherbedarf für das Array C der Länge n ist $\Theta(n)$. Falls man zusätzlich A und B als Eingaben betrachtet, dann benötigt man insgesamt $\Theta(n)$ Speicher zusätzlich zu den Eingaben (also insgesamt $\Theta(n)$ Speicher für die drei Arrays zusammen, da A, B bereits vorliegen). Somit ist der Speicherbedarf für C selbst $\Theta(n)$ und der Gesamtspeicheraufwand inkl. Eingaben ebenfalls $\Theta(n)$.

(c) Lösung: Das dynamische Array verdoppelt seine Kapazität bei Bedarf. Die amortisierte Laufzeit pro Einfügung ist $\mathcal{O}(1)$. Begründung: In n Einfügungen werden höchstens $2n - 1$ Kopien von Elementen durchgeführt (bis zur Verdopplung der Kapazität), weshalb die Gesamtkosten $\mathcal{O}(n)$ betragen. Teufelchen der Kostenproportionen wird somit durch amortisierte Kosten pro Einfügung zu konstanten Kosten.

(d) Lösung: Zwei Vorteile von Hash-Tabellen gegenüber Arrays bei Schlüsselzugriffen: - Erwartete konstante Laufzeit für Lookup und Insert: $\mathcal{O}(1)$ im Durchschnitt, unabhängig von der Domaingröße. - Dynamische Größenausdehnung: Hash-Tabellen können flexibel wachsen, ohne dass eine feste, vorgegebene Reihenfolge der Keys nötig ist.

Eine einfache Hash-Funktion (für ganzzahlige Schlüssel) ist z. B.

$$h(k) \;=\; k \bmod m,$$

wobei m die Anzahl der Buckets (Bucketszahl) ist. Bei Kollisionsauflösung (z. B. offener Adressierung oder separate Verkettung) funktioniert diese Funktion zuverlässig.

(a) Lösung: - Anzahl der Kanten $|E| = 5$. - Knotengrade: $\deg(1) = 2$ (Kanten (1,2),(1,3)), $\deg(2) = 3$ (Kanten (1,2),(2,3),(2,4)), $\deg(3) = 3$ (Kanten (1,3),(2,3),(3,5)), $\deg(4) = 1$ (Kante (2,4)), $\deg(5) = 1$ (Kante (3,5)).

(b) Lösung: BFS beginnt bei Startknoten 1; bei sortierten Nachbarschaften (nach Ziffern) ergibt sich folgende Besuchsreihenfolge:

$$1,\ 2,\ 3,\ 4,\ 5.$$

Dies ergibt sich aus der Queuierung der unbesuchten Nachbarn in der Reihenfolge 2, 3 (von 1), dann 4 (von 2) und 5 (von 3).

(c) Lösung: Gewichteter Graph (unter der Annahme, dass $V = \{s, u, v, w, t\}$ und $E = \{(s,u), (s,v), (u,w), (v,w), (w,t)\}$): - Pfade und Kosten: - $s \to u \to w \to t$: $10 + 15 + 7 = 32$ - $s \to v \to w \to t$: $5 + 4 + 7 = 16$

- Nach Dijkstra ist der kürzeste Pfad von s nach t der Pfad

$$s \to v \to w \to t$$

mit Distanz 16. Die Zwischendistanzen (skizziert) wären: $\text{dist}(s) = 0$, $\text{dist}(v) = 5$, $\text{dist}(u) = 10$ (via $s \to u$), $\text{dist}(w) = 9$ (via v), $\text{dist}(t) = 16$ (via w).

(d) Lösung: Für sehr dichte Graphen eignet sich eine Adjazenzmatrix besser. Begründung: - Bei dichten Graphen ($m \approx n^2$) hat die Matrix denselben asymptotischen Speicheraufwand wie der Graph durch Kanten, aber der Zugriff auf den Nachbarn eines Knotens (Nachbarschaftsrundgang) ist oft konstanter bzw. deterministisch einfach über die Matrix abzuleiten. - Adjazenzlisten sparen Speicher nur bei sparsamen Graphen; bei dichter Graphdichte führt das Listenformat über viele Kanten zu vielen Pointer-Overheads, während die Matrix eine feste Zugriffskostenstruktur bietet.

(a) Lösung: Maximalflusswert = 20. Ein akzeptierendes Min-Cut-Beispiel ist

$$S = \{s, a, b\}, \quad T = \{t\}.$$

Die Kapazität des Cuts ist $c(S, T) = w(b, t) = 20$. Da der Fluss diesen Wert erreicht, ist dies ein Min-Cut und der Max-Flow entspricht 20.

(b) Lösung: Das Max-Flow-Min-Cut-Theorem besagt: Der maximale Fluss von s nach t in einem Flussnetzwerk ist gleich der Kapazität des kleinsten s-t-Cuts in diesem Graphen. Mit anderen Worten: $\max \text{Flow}(s, t) = \min_{(S,T)} c(S, T)$, wobei S ein Teilmenge der Knoten enthält, die s enthält und t nicht.

(c) Lösung: Max-Flow-Konzepte lassen sich auf Scheduling-Probleme übertragen, indem man eine zeitabhängige Netzwerkstruktur nutzt (Zeit-Expanded Network). Typischer Aufbau: - Erzeuge Kopien der Ressourcen/Zeitpunkte als Knoten; Verbindungen zwischen aufeinanderfolgenden Zeitpunkten modellieren die Verfügbarkeit einer Ressource. - Kantenkapazitäten repräsentieren maximale Auslastungen pro Zeitscheibe bzw. erlaubte Zuweisungen. - Eine maximale Flussinjektion entspricht der maximal möglichen Anzahl (oder Gesamtmenge) von Aufgaben, die innerhalb der vorgesehenen Zeitspanne erledigt werden können. - Ein Min-Cut entspricht subsystematischen Engpässen (Ressourcen- oder Zeitbeschränkungen), deren Brechen den Durchsatz begrenzt.

Kurz gesagt: Scheduling-Probleme lassen sich durch geeignete Zeit-Expanded-Netzwerke in Max-Flow-Modelle überführen; der maximale Fluss entspricht der maximalen ausführbaren Arbeit, und Min-Cuts liefern optimale Engpass-Restriktionen.

(a) Lösung: Grundoperationen einer binär-Heap-basierten Prioritätswarteschlange: - Insert(x): Einfügen von x in den Heap; Laufzeit Worst-Case $\mathcal{O}(\log n)$. - Extract-Min(): Entfernen des kleinsten Elements (Wurzel) und anschließendes Re-Heapifizieren; Laufzeit Worst-Case $\mathcal{O}(\log n)$.

(b) Lösung: - Heapsort ist in der Regel inplace und verwendet $\mathcal{O}(1)$ zusätzlicher Speicher, ist aber nicht stabil. - Mergesort ist stabil (gleiche Schlüssel behalten relative Reihenfolge) und benötigt typischerweise $\mathcal{O}(n)$ zusätzlichen Speicher (bei der klassischen Implementierung). Es existieren auch in-place-Varianten, die jedoch komplexer sind.

(c) Lösung: Zähle Inversionspaare mit einem modifizierten Mergesort (Inversionszählung im Merge-Schritt): - Teile das Array rekursiv in zwei Hälften. - Beim Zusammenführen zählt man Cross-Inversions: Wenn das linke Element $a[i]$ größer als das rechte $a[j]$ ist, so ergeben sich $\text{LeftSize} - i + 1$ Inversionen (alle verbleibenden Elemente der linken Hälfte sind größer als $a[j]$).

- Die Gesamtanzahl der Inversionen ist die Summe der Inversionen aus den rekursiven Teilen plus den Inversionen beim Merge. - Laufzeit: $\mathcal{O}(n \log n)$; Speicherbedarf: $\mathcal{O}(n)$ zusätzlich für das Merge-Array (bei der Standardimplementierung).