Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Roman Böhringer

February 8, 2018

Data Structures and Algorithms Zusammenfassung

1 Asymptotische Notation

1.1 *Obere Schranke*

$$O(f) \leq O(g) \Leftrightarrow \exists c, n_0 \forall n \geq n_0 (f(n) \leq c * g(n))$$

1.2 *Untere Schranke*

$$\Omega(f) \geq \Omega(g) \Leftrightarrow \exists c, n_0 \forall n \geq n_0 (f(n) \geq c * g(n))$$

1.3 *Scharfe Schranke*

$$\Theta(f) = \Theta(g) \Leftrightarrow O(f) \leq O(g) \text{ and } \Omega(f) \geq \Omega(g)$$

2 Rechenregeln

2.1 *Gausssche Summenformel*

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

2.2 *Summe der ersten n Quadratzahlen*

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

3 Sortieren

3.1 *Bubblesort*

Es wird jeweils geprüft, ob $A[i] > A[i+1]$. Ist dies der Fall, werden sie vertauscht. Die Laufzeit beträgt $\Theta(n^2)$.

3.2 *Sortieren durch Auswahl (Selectionsort)*

Es wird der maximale Schlüssel zwischen $A[k]$ und $A[2]$ ermittelt und mit der Position $A[k]$ vertauscht. Somit weniger Schlüsselvertauschungen als bei Bubblesort, aber die gleiche Laufzeit von $\Theta(n^2)$.

3.3 *Sortieren durch Einfügen (Insertionsort)*

Bei Insertionsort ist die Induktionsannahme, dass ein gewisser Teil des Arrays bereits aufsteigend sortiert ist, jedoch noch nicht zwingend die k kleinsten Schlüssel enthält. Somit benutzt man binäre Suche auf $A[1]$ bis $A[k]$, um die Position von $A[k+1]$ zu finden und fügt $A[k+1]$ an dieser Position ein.
Mit binärer Suche beträgt die Laufzeit des Algorithmus $O(n \log n)$, ohne binäre Suche beträgt sie im Worst Case $O(n^2)$.

3.4 *Heapsort*

Bei Heapsort wird die Eigenschaft eines Heaps genutzt, dass in einer Laufzeit von $O(\log n)$ das Maximum extrahiert werden kann und die Heap Condition wiederhergestellt werden kann. Somit wird $n-1$ mal das Maximum extrahiert und die Heap Condition wiederhergestellt, weswegen die Gesamtlaufzeit $O(n \log n)$ beträgt.

3.5 *Mergesort*

Beim Mergesort wird ein Array so lange rekursiv aufgesplittet, bis es nur noch ein Element enthält. Die aufgesplitteten Arrays werden dann zusammengeführt, wobei jeweils der Pointer i auf das Element $left$ zeigt und der Pointer j auf das Element $right$. $A[i]$ und $A[j]$ werden jeweils miteinander verglichen und das kleinere Element landet im Zielarray. Wenn $i == mid$ oder $j == right$ ist, werden die restlichen Elemente hinten angehängt. Die Laufzeit beträgt $O(n \times \log n)$.

3.6 *Quicksort*

Bei Quicksort wird ein Pivotelement p gewählt und die Elemente werden anhand von diesem Element aufgesplittet. Die kleineren Elemente landen links vom Pivot, die grösseren Elemente landen rechts vom Pivot. Anschliessend werden die beiden Teile rekursiv mit Quicksort sortiert. Eine Beispielimplementation von Quicksort in Java sieht folgendermassen aus (wobei das mittlere Element als Pivot gewählt wird):

```
public static void quickSort(int[] arr, int low, int high) {
  if (arr == null || arr.length == 0)
     return;

  if (low >= high)
     return;

  // pick the pivot
  int middle = low + (high - low) / 2;
  int pivot = arr[middle];

  // make left < pivot and right > pivot
  int i = low, j = high;
  while (i <= j) {
     while (arr[i] < pivot) {
        i++;
     }

     while (arr[j] > pivot) {
        j--;
     }
```

```
        if (i <= j) {
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            i++;
            j--;
        }
    }

    // recursively sort two sub parts
    if (low < j)
        quickSort(arr, low, j);

    if (high > i)
        quickSort(arr, i, high);
}
```

3.7 *Übersicht*

Algorithmus	**Worst case Laufzeit**	**Best case Laufzeit**	**In place**	**Stabil**
Bubblesort	$O(n^2)$	$O(n)$	Ja	Ja
Selectionsort	$O(n^2)$	$O(n^2)$	Nein	Grundsätzlich nicht, lässt sich stabil implementieren
Insertionsort	$O(n \log n)$ (mit binärer Suche)	$O(n)$	Ja	Ja
Heapsort	$O(n \log n)$	$O(n \log n)$	Ja	Nein
Mergesort	$O(n \log n)$	$O(n \log n)$	Normalerweise nicht	Ja
Quicksort	$O(n^2)$ (jedoch sehr unwahrscheinlich)	$O(n \log n)$	Ja	Nein

3.8 *Untere Schranke für vergleichbasiertes Sortieren*

Bei vergleichsbasiertem Sortieren haben wir einen Entscheidungsbaum mit $n!$ Knoten (jede Permutation der Eingabewerte). Die Höhe des Baumes beträgt somit $log_2(n)$. Es ist einfach zu sehen, dass $n! \geq (n/2)^{n/2}$. Somit ist $\log(n!) \geq \log((n/2)^{n/2}) = n/2 * \log(n/2)$, womit die untere Schranke $\Omega(n \log(n))$ ist.

4 Datenstrukturen

4.1 *Natürliche Suchbäume*

Ein binärer Suchbaum oder natürlicher Suchbaum erfüllt die Suchbaumeigenschaft: Für jeden Knoten v gilt, dass alle im linken Teilbaum vom v gespeicherten Schlüssel kleiner als der Schlüssel

von v sind und alle im rechten Teilbaum gespeicherten Schlüssel grösser als v sind. Für das Suchen und Einfügen muss der Baum von der Wurzel aus traversiert werden, was in Laufzeit $O(h)$ (wobei h die Höhe ist) geschieht.
Beim Löschen eines Knotens ist der Fall interessant, wenn beide Nachfolger Teilbäume sind (bei nur einem Teilbaum als Nachfolger kann dieser angehängt werden, bei keinem kann der Knoten einfach gelöscht werden). In diesem Fall gibt es einen symmetrischen Nachfolger und Vorgänger: Der grösste Knoten des linken Teilbaums oder der kleinste Knoten des rechten Teilbaums. Wird der gelöschte Knoten mit diesem Nachfolger / Vorgänger ersetzt, bleibt die Suchbaumeigenschaft erhalten.
Zwei binäre Suchbäume, bei welchen der maximale Wert eines Schlüssels in K_1 kleiner als der minimale Wert in K_2 ist, können einfach vereinigt werden. Dazu wird der minimale Knoten aus K_2 ermittelt und als neue Wurzel verwendet. Der linke Teilbaum ist K_1, der rechte Teilbaum K_2 (ohne den minimalen Knoten).
Es gibt drei Durchlaufordnungen für Bäume:

- Hauptreihenfolge (Preorder): Zunächst wird v ausgegeben, dann $T_l(v)$ und dann $T_r(v)$.
- Nebenreihenfolge (Postorder): Zuerst wird $T_l(v)$, dann $T_r(v)$ und dann wird v ausgegeben.
- symmetrische Reihenfolge (Inorder): Zunächst wird $T_l(v)$ rekursiv besucht, dann v ausgegeben und dann $T_r(v)$ rekursiv besucht. Die Knoten werden somit in aufsteigend sortierter Reihenfolge ausgegeben!

Anhand der Pre- und Postorder Ausgabe kann der Baum rekonstruiert werden (man ordnet rekursiv für alle Teilbäume v so ein, dass die Suchbaumeigenschaft erfüllt ist).

4.2 *AVL-Bäume*

Bei natürlichen Suchbäumen ist die Höhe nicht beschränkt und kann im schlechtesten Fall (sortierte Werte beim Einfügen) linear werden. Deswegen wird für AVL-Bäume die Balance $bal(v)$ für jeden Knoten eingeführt. Sie ist wie folgt definiert:

$$bal(v) := h(T_r(v)) - h(T_l(v))$$

Für AVL-Bäume gilt zusätzlich die AVL-Bedingung, die besagt, dass für alle Knoten $bal(v) \in \{-1, 0, 1\}$ gilt.
Mittels Induktion kann bewiesen werden, dass die Höhe von AVL-Bäumen durch $\log n$ beschränkt ist. Dazu untersucht man, wie viele Blätter ("Nullpointer") ein AVL-Baum mit einer gegebenen Höhe mindestens hat. Denn es gilt, dass jeder binäre Suchbaum mit n Knoten genau $n+1$ Blätter hat. Die Induktionsverankerungen sind $MB(1) = 2$ und $MB(2) = 3$. Für $h \geq 3$ betrachtet man die Mindestblattzahl des linken Teilbaums und des rechten Teilbaums. Ein Teilbaum muss mindestens Höhe $h-1$ haben, der andere wegen der AVL-Bedingung $h-1$ oder $h-2$. Also gilt die Formel: $MB(h) = MB(h-1) + MB(h-2)$. Dies ist sehr ähnlich zu den Fibonacci-Zahlen, es gilt nämlich: $MB(h) = F_{h+2}$. Somit kann gezeigt werden, dass ein AVL-Baum mit Höhe h mindestens 1.6^h Blätter und somit mindestens $1.6^h - 1$ Knoten hat. Die Höhe eines AVL-Baums mit n Schlüsseln ist also durch $1.44 \log_2 n$ nach oben beschränkt.

4.3 *Heap*

Ein Max-Heap (dasselbe gilt analog für einen Min-Heap) hat die Eigenschaft, dass alle Schlüssel "unter" ihm grösser sind. Wenn das Maximum extrahiert wurde, wird das Element $A[m]$ an die Position $A[1]$ verschoben und so lange mit dem grösseren der beiden Nachfolger getauscht, bis

dies nicht mehr möglich ist. Um den Heap zu erzeugen, wird diese Methode (Restore-Heap-Condition) für die Positionen $\lceil \frac{n}{2} \rceil$ aufgerufen, womit der Heap in Laufzeit $O(n)$ erstellt werden kann.

5 Verschiedene Algorithmen

5.1 *Stable Marriage Problem (SMP)*

Problem:

1. n Männer und n Frauen.
2. Jede Frau f_i hat eine Rangliste der Männer.
3. Jeder Mann m_i hat eine Rangliste der Frauen.

Gesucht ist eine stabile Zuordnung. Eine Zuordnung ist eine Menge von Paaren (m_i, f_i), so dass keine Frau und kein Mann mehr als einmal vorkommt. Ein Paar (m_i, f_j) ist stabil für eine Zuordnung, wenn entweder m_i seine aktuelle Partnerin gegenüber f_j vorzieht oder f_j ihren aktuellen Partner gegenüber m_i vorzieht. Die Zuordnung ist stabil, wenn alle möglichen Paare (m_i, f_i) stabil sind.

5.1.1 Gale-Shapley Algorithmus

Prinzipien:

1. Männer sind aktiv und stellen Anträge
2. Frauen sind passiv und akzeptieren Anträge / lehnen diese ab.
3. Beide sind eigennützig, sie stellen Anträge und reagieren auf Anträge gemäss ihrer Präferenzen
4. Beide sind flexibel, zuvor akzeptierte Anträge können wieder aufgelöst werden.

Die Rollen können dabei getauscht werden, wichtig ist nur die Asymmetrie zwischen den beiden Seiten.
Algorithmus:

1. beginne mit einer leeren Zuordnung
2. solange es einen Mann m gibt, der keinen Partner hat und noch nicht jeder Frau einen Antrag gestellt hat:
   a) m_i stellt einen Antrag an die Frau f_j , die er bevorzugt unter allen, die er noch nicht gefragt hat
   b) f_j akzeptiert den Antrag, falls sie noch keinen Partner hat oder sie m_i vorzieht gegenüber ihrem aktuellen Partner (in diesem Fall löst sie die bestehende Partnerschaft auf)

Da jeder Mann höchstens n Anträge stellt, gibt es höchstens n^2 Anträge.
Der Algorithmus ist Mann-optimal und Frau-pessimal (jeder Mann hat den bestmöglichen Partner) und jede Frau hat den schlechtmöglichen Partner.

5.2 *Maximum Subarray*

Gegeben ist ein Array mit n Zahlen. Gesucht ist ein zusammenhängedes Teilstück mit der maximalen Summe. Das Problem kann in einer Laufzeit von $O(n)$ gelöst werden. Dazu wird sich eine Summe "randmax" gemerkt, in welcher notiert ist, wie die beste Lösung bis a_i aussieht. Wenn der Wert unter 0 ist, wird "randmax" auf 0 zurückgesetzt. Wenn randmax grösser als die aktuell maximale Summe ist, wird diese auf randmax gesetzt.

5.3 *Längste aufsteigende Teilfolge*

Es wird sich für jede Folgenlänge die kleinste End-zahl einer längsten aufsteigenden Teilfolge gemerkt und dann geprüfut, ob diese verlängert werden kann / mit einem kleineren letzten Element ersetzt werden kann.

5.4 *Längste gemeinsame Teilfolge*

$$L(m,n) \leftarrow \max\left(L(m-1,n-1)+\delta_{mn}, L(m,n-1), L(m-1,n)\right)$$

Wobei $\delta_{mn} = 1$ ist, wenn $a_m = b_n$.

5.5 *Editierdistanz*

Wie viele Veränderungen (Entfernen, hinzufügen, ändern) bracuht es, um von einem Wort auf ein anderes zu kommen? Es soll ein möglichst kurzes Alignment erstellt werden. Dazu kann ein Buchstabe jedes Wortes weggelassen werden oder beide:

$$ED(n,m) = \min \begin{cases} ED(n-1,m)+1 \\ ED(n,m-1)+1 \\ ED(n-1,m-1)+d \text{ d=1, falls } a_n \neq b_m;\ d=0 \text{ sonst} \end{cases}$$

5.6 *Matrixkettenmultiplikation*

$$M(i,j) = \min_k \left\{ \underbrace{M(i,k)+M(k+1,j)}_{\text{optimale Teilstücke}} + \underbrace{u \cdot v \cdot w}_{\text{letzte Multiplikation}} \right\}$$

5.7 *Rucksackproblem / Rundreisenproblem*

Gegeben sind N Gegenstände, wobei Gegenstand i das Gewicht g_i und den Wert w_i hat sowie ein Rucksack, der Gesamtgewicht G tragen kann. Es werden die Gegenstände mit möglichst hohem Gesamtwert gesucht. Das Problem kann pseudopolynomiell mit einer DP-Tabelle der Grösse NG in Laufzeit $O(NG)$ gelöst werden. Ein Eintrag berechnet sich dabei wie folgt:

$$maxwert[i,g] = \max(maxwert[i-1,g], w_i + maxwert[i-1, g-g_i])$$

6 Graphenalgorithmen

6.1 *Topologische Sortierung*

Eine topologische Sortierung ist eine Sequenz der Knoten, so dass die "Reihenfolge" der Kanten berücksichtigt wird. Ein gerichteter Graph besitzt genau dann eine topologische Sortierung, wenn er kreisfrei ist.

Zur Bestimmung der topologischen Sortierung kann eine modifizierte Tiefensuche verwendet werden. Dabei müssen sich jeweils die aktiven Knoten gemerkt werden (falls man nochmals auf sie stosst, hat der Graph einen Zyklus und es existiert keine topologische Sortierung) und die Knoten müssen zur Lösung hinzugefügt werden, sobald sie verlassen werden.

6.2 *Tiefensuche*

Bei der Tiefensuche wird jeder Nachbar direkt besucht, bis man bei einem bereits besuchten Knoten ankommt. Dann wird "zurückgegangen" und der nächste unbesuchte Knoten besucht. Die Laufzeit beträgt $O(|V| + |E|)$.
Die Tiefensuche kann rekursiv oder iterativ mit einem Stack implementiert werden. Die iterative Implementierung sieht folgendermassen aus:

$S \leftarrow \emptyset$
PUSH(v, S)
while $S \neq \emptyset$ **do**
  $w \leftarrow$ POP(S)
  if w not visited **then**
    Mark w as visited
    for each (w, x) **do**
      if x not visited **then** PUSH(x, S)

Die Tiefensuche kann auch zum Herausfinden der Zusammenhangskomponenten benutzt werden (jedes mal, wenn neu "angesetzt" wird handelt es sich um eine neue Zusammenhangskomponente).

6.3 *Breitensuche*

Bei der Breitensuche werden zuerst alle Nachfolger eines Knotens, dann alle Nachfolger der Nachfolger, etc... besucht. Die Breitensuche kann gleich wie die Tiefensuche implementiert werden, jedoch wird eine Queue als Datenstruktur verwendet.

6.4 *Reflexive, transitive Hülle*

Die reflexive, transitive Hülle ist die Erreichbarkeitsrelation eines Graphens.

6.5 *Kürzeste Wege*

6.5.1 Uniforme Kantengewichte

Bei uniformen Kantengewichten kann eine modifizierte Breitensuche verwendet werden. Dabei wird sich jeweils der Vorgänger gemerkt. Somit hat man sogar die kürzesten Wege des Startknotens zu allen anderen Knoten.

6.5.2 Nicht-negative Kantengewichte - Dijkstras Algorithmus

Bei Dijkstras Algorithmus wird sich die Länge des kürzesten Weges zu jedem Knoten gemerkt (und der Vorgänger). Dann wird über alle Knoten iteriert (wie in der Breitensuche) und jeweils $d[u]$ aktualisiert, wenn eine Verbesserung möglich ist. Es wird jeweils der unbesuchte Knoten mit dem minimalen $d[v]$ iteriert und weiter gemacht.

6.5.3 Allgemeine Kantengewichte - Bellmann-Ford Algorithmus

Der Algorithmus von Bellmann-Ford ist ein DP-Algorithmus mit einer Tabelle der Grösse $|V| \times |V|$. Ein Eintrag berechnet sich wie folgt:

$$T[v,i] \leftarrow \min\Big(T[v][i-1], \min_{(u,v)\in E}(T[u][i-1] + w((u,v))\Big)$$

Es wird also geschaut, ob in der vorherigen Iteration bereits das Minimum gefunden wurde oder ob mit einer anderen Kante ein kürzerer Weg gefunden werden kann.
Am Schluss wird noch geprüft, ob die Werte für $d[v]$ (welche sich in $T[v, n-1]$ befinden) mit irgendeinem $d[u] + w((u,v))$ verbessert werden können, in welchem Fall der Graph einen negativen Zyklus enthält.

6.6 *Minimale Spannbäume*

Ein minimaler Spannbaum eines Graphen ist ein Spannbaum von G (alle Knoten werden verbunden) mit minimaler Gesamtlänge.

6.6.1 Kruskal

Anfangs ist jeder einzelne Knoten des Graphen ein Baum. In aufsteigender Reihenfolge der Kantenlängen wird dann folgende Auswahlregel angewandt: Falls e beide Endknoten im selben gewählten Baum hat, verwirf e, sonst wähle e.