FIRST DRAFT - created using Gemini

0.1 Suchen und Sortieren

0.1.1 Suchen

Diese Vorlesung beschäftigt sich mit zwei fundamentalen Aufgaben der Informatik: Suchen und Sortieren. Viele der Berechnungen, die in Rechenzentren durchgeführt werden, beschäftigen sich mit diesen Problemen – dementsprechend wichtig ist es, effiziente Algorithmen zu finden.

Beispiel: Finde die Prüfung von Maria Miller in einem Stapel aus 600 Prüfungen.
Formal betrachten wir das folgende Problem:

Definition 0.1 (Suche)**.** *Gegeben seien ein Array A mit n Einträgen (bei uns: Zahlen) und ein Element b. Gesucht ist ein Index k mit $A[k] = b$, oder die Rückgabe "nicht gefunden", falls b nicht in A enthalten ist.*

Wir betrachten zwei Fälle:

- Fall 1: A ist unsortiert
- Fall 2: A ist sortiert, d.h., $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Fall 1: A ist unsortiert

In diesem Fall machen wir keinerlei Annahmen über das Array A. Der einfachste Algorithmus zur Lösung des Suchproblems in diesem Fall ist die lineare Suche:

LINEAR-SEARCH$(A = (A[1], \ldots, A[n]), b)$ — LINEARE SUCHE

```
1 for i ← 1, 2, ..., n do
2     if A[i] = b then return i
3 return "nicht gefunden"
```

Die Laufzeit der linearen Suche beträgt im schlechtesten Fall offensichtlich $\Theta(n)$, denn die Schleife wird maximal n Mal durchlaufen. Tatsächlich ist die lineare Suche optimal, d.h., im schlechtesten Fall muss *jeder* Suchalgorithmus n viele Vergleiche durchführen. Um dies zu begründen, stellen wir uns vor, dass b nicht in A enthalten ist. Nachdem der Algorithmus $n-1$ viele Elemente von A gelesen hat, kann er noch nicht entscheiden, ob b in A enthalten ist, oder nicht (denn b könnte ja im letzten, noch nicht gelesenen Element von A gespeichert sein). — LAUFZEIT

FINAL VERSION

KAPITEL 1

Suchen

Dieses Kapitel beschäftigt sich mit zwei fundamentalen Aufgaben der Informatik: Suchen und Sortieren. Viele der Berechnungen, die in Rechenzentren durchgeführt werden, beschäftigen sich mit diesen Problemen – dementsprechend wichtig ist es, effiziente Algorithmen zu finden.

Beispiel: Finde die Prüfung von Maria Müller in einem Stapel aus 600 Prüfungen.
Formal betrachten wir das folgende Problem:

Definition 1.1 (Suche)**.** *Gegeben seien ein Array A mit n Einträgen (bei uns: Zahlen) und ein Element b. Gesucht ist ein Index k mit $A[k] = b$, oder die Rückgabe "nicht gefunden", falls b nicht in A enthalten ist.*

Die Annahme, dass das Array mit Zahlen gefüllt ist, ist keine Einschränkung. Im Computer ist ja jedes Objekt durch eine Folge von Nullen und Einsen gegeben. Diese kann man immer auch als Zahl interpretieren.

Wir betrachten zwei Fälle:

- Fall 1: A ist unsortiert
- Fall 2: A ist sortiert, d.h., $A[1] \leq A[2] \leq \cdots \leq A[n]$.

1.1 Unsortierte Arrays

In diesem Fall machen wir keinerlei Annahmen über das Array A. Der einfachste Algorithmus zur Lösung des Suchproblems in diesem Fall ist die lineare Suche:

LINEAR-SEARCH$(A[1..n], b)$ — LINEARE SUCHE

```
1 for i ← 1, 2, ..., n do
2     if A[i] = b then return i
3 return "nicht gefunden"
```

Die Laufzeit der linearen Suche beträgt im schlechtesten Fall (worst case) offensichtlich $\Theta(n)$, denn die Schleife wird maximal n Mal durchlaufen. Tatsächlich ist die lineare Suche optimal, d.h., im schlechtesten Fall muss *jeder* Suchalgorithmus n viele Vergleiche durchführen. Um dies zu begründen, stellen wir uns vor, dass b nicht in A enthalten ist. Nachdem der Algorithmus $n-1$ viele Elemente von A gelesen hat, kann er noch nicht entscheiden, ob b in A enthalten ist, oder nicht (denn b könnte ja im letzten, noch nicht gelesenen Element von A gespeichert sein). — LAUFZEIT

Fall 2: A ist sortiert

Wir nehmen nun an, dass A sortiert ist, d.h., $A[1] \leq A[2] \leq \cdots \leq A[n]$. Die lineare Suche funktioniert natürlich auch in diesem Fall, doch wir können die zusätzliche Information, dass A sortiert ist, ausnutzen, um deutlich schneller zu sein.

Ist A sortiert, dann können wir das Suchproblem mittels Divide-and-Conquer lösen. Dazu betrachten wir die mittlere Position $m = \lfloor n/2 \rfloor$ im Array und unterscheiden drei Fälle:

1) Ist $b = A[m]$, dann haben wir den Schlüssel b an Position m gefunden und geben daher m zurück.

2) Ist $b < A[m]$, dann suchen wir rekursiv in der linken Hälfte (also auf den Positionen $1, \ldots, m-1$) weiter nach b.

3) Ist $b > A[m]$, dann suchen wir rekursiv in der rechten Hälfte (also auf den Positionen $m+1, \ldots, n$) weiter nach b.

KORREKTHEIT Dieses Verfahren wird *binäre Suche* genannt. Die Korrektheit der binären Suche folgt direkt aus der Tatsache, dass A aufsteigend sortiert ist. Ist also $b < A[m]$, dann wissen wir bereits, dass die Positionen $m, \ldots, n$ nur Schlüssel enthalten, die grösser als b sind. Also reicht es, die Suche auf die ersten $m-1$ Positionen einzuschränken. Die Begründung für den Fall $b > A[m]$ verläuft analog. Da der Suchbereich (die Anzahl der noch möglichen Positionen) in jedem Schritt um mindestens ein Element kleiner wird, endet das Verfahren auch nach einer endlichen Zahl von Schritten.

In der Realität würde man zudem die binäre Suche iterativ statt rekursiv implementieren, was nicht schwierig ist, wie der folgende Pseudocode zeigt.

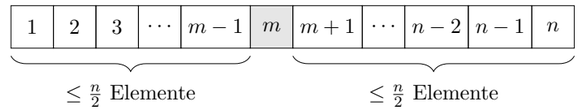
BINÄRE SUCHE BINARY-SEARCH$(A = (A[1], \ldots, A[n]), b)$

```
1 left ← 1; right ← n                                  ▷ Initialer Suchbereich
2 while left ≤ right do
3     middle ← ⌊(left+right)/2⌋
4     if A[middle] = b then return middle              ▷ Element gefunden
5     else if A[middle] > b then right ← middle−1      ▷ Suche links weiter
6     else left ← middle+1                             ▷ Suche rechts weiter
7 return "Nicht vorhanden"
```


1.2 Sortierte Arrays: Binäre Suche

Wir nehmen nun an, dass A sortiert ist, d.h., $A[1] \leq A[2] \leq \cdots \leq A[n]$. Die lineare Suche funktioniert natürlich auch in diesem Fall, doch wir können die zusätzliche Information, dass A sortiert ist, ausnutzen, um deutlich schneller zu sein.

Ist A sortiert, dann können wir das Suchproblem mittels Divide-and-Conquer lösen. Dazu betrachten wir die mittlere Position $m = \lfloor (n+1)/2 \rfloor$ im Array und unterscheiden drei Fälle:



1) Ist $b = A[m]$, dann haben wir den Schlüssel b an Position m gefunden und geben daher m zurück.

2) Ist $b < A[m]$, dann suchen wir rekursiv in der linken Hälfte (also auf den Positionen $1, \ldots, m-1$) weiter nach b.

3) Ist $b > A[m]$, dann suchen wir rekursiv in der rechten Hälfte (also auf den Positionen $m+1, \ldots, n$) weiter nach b.

KORREKTHEIT Dieses Verfahren wird *binäre Suche* genannt. Die Korrektheit der binären Suche folgt direkt aus der Tatsache, dass A aufsteigend sortiert ist. Ist also $b < A[m]$, dann wissen wir bereits, dass die Positionen $m, \ldots, n$ nur Schlüssel enthalten, die grösser als b sind. Also reicht es, die Suche auf die ersten $m-1$ Positionen einzuschränken. Die Begründung für den Fall $b > A[m]$ verläuft analog. Da der Suchbereich (die Anzahl der noch möglichen Positionen) in jedem Schritt um mindestens ein Element kleiner wird, endet das Verfahren auch nach einer endlichen Zahl von Schritten.

Binäre Suche kann wie alle rekursiven Algorithmen entweder rekursiv oder iterativ implementiert werden. Wir geben hier einen Pseudocode mit der Invariante[1], dass $A[\ell..r]$ der verbleibende Suchbereich ist.

BINÄRE SUCHE BINARY-SEARCH$(A[1..n], b)$

```
1 ℓ ← 1; r ← n                                         ▷ Initialer Suchbereich
2 while ℓ ≤ r do
3     m ← ⌊(ℓ + r)/2⌋
4     if A[m] = b then return m                        ▷ Element gefunden
5     else if A[m] > b then r ← m − 1                  ▷ Suche links weiter
6     else ℓ ← m + 1                                   ▷ Suche rechts weiter
7 return "Nicht vorhanden"
```

[1] Das Konzept der Invariante behandeln wir später noch ausführlicher.

Wie viele Schritte sind dies im schlimmsten Fall? Dieser tritt offenbar dann ein, wenn erfolglos nach einem Schlüssel gesucht wird. Für ein Array der Länge $n = 2^k$ ergibt sich die folgende Rekurrenz zur Abschätzung der maximal durchgeführten Operationen bei einem Array A der Länge n:

LAUFZEIT

$$T(n) = \begin{cases} c & \text{falls } n = 0 \text{ ist,} \\ T(n/2) + d & \text{falls } n \geq 1 \text{ ist,} \end{cases} \tag{1}$$

wobei c und d jeweils konstant sind. Mit den früher in der Vorlesung vorgestellten Methoden erhalten wir $T(n) \in \mathcal{O}(\log n)$, was sehr schnell ist.

Wie im unsortierten Fall stellt sich auch hier die Frage, ob es besser geht. Die Antwort ist wieder nein – auch wenn der Beweis etwas aufwändiger ist.

Wie viele Schritte $T(n)$ sind dies im schlimmsten Fall? Dazu beobachten wir, dass die Zahl der verbleibenden Positionen sowohl in Fall 2) als auch in Fall 3) höchstens $n/2$ ist. Da die Laufzeit für längere Arrays höchstens länger sein kann[2], müssen wir nach Absteigen in die Rekursion noch höchstens Zeit $T(n/2)$ aufbringen. Für ein Array der Länge $n = 2^k$ ergibt sich damit die folgende Rekurrenz zur Abschätzung der maximalen Zahl $T(n)$ an durchgeführten Operationen:

LAUFZEIT

$$T(n) \leq \begin{cases} c & \text{falls } n = 1 \text{ ist,} \\ T(n/2) + d & \text{falls } n \geq 2 \text{ ist,} \end{cases} \tag{1}$$

wobei c und d jeweils konstant sind.

Wir zeigen nun per vollständiger Induktion, dass $T(n) \leq d \log_2(n) + c$ für alle $n = 2^k$ mit $k \geq 0$ ist.

Anfang: Für $k = 0$ folgt die Behauptung direkt aus (1).

$k \to k+1$: Für $n = 2^{k+1}$ schätzen wir ab:

$$\begin{aligned} T(n) &\leq T(n/2) + d && (1) \\ &= T(2^k) + d \\ &\leq d \log_2(2^k) + c + d && \text{Induktionshypothese für } n = 2^k \\ &= d \cdot (k+1) + c \\ &= d \log_2(n) + c. \end{aligned}$$

Damit ist der Induktionsschritt gezeigt.

Damit folgt, dass $T(n) \leq O(\log n)$, was sehr schnell ist. Wie im unsortierten Fall stellt sich auch hier die Frage, ob es besser geht. Die Antwort ist wieder nein – auch wenn der Beweis etwas aufwändiger ist.

[2] Diese Eigenschaft kann man formal mit Induktion beweisen. Das werden wir hier aber nicht tun.

Abb. 1 Binäre Suche auf einem Array der Länge 6 als Entscheidungsbaum visualisiert. Da sich das zu suchende Element im Erfolgsfall auf sechs möglichen Positionen befinden kann, hat der Entscheidungsbaum die gleiche Anzahl von Knoten. Im ersten Schritt wird $A[3]$ mit b verglichen. Bei Gleichheit ist der Algorithmus fertig, ansonsten werden $A[1]$ bzw. $A[5]$ verglichen, usw. Da der Baum Höhe 3 hat (also aus drei "Schichten" besteht), ist die binäre Suche auf einem Array mit sechs Elementen nach spätestens drei Vergleichen fertig.

Untere Schranke

Untere Schranke Um zu beweisen, dass die Suche in sortierten Arrays im schlimmsten Fall immer $\Omega(\log n)$ viele Vergleiche ausführt, betrachten wir wieder einen *beliebigen* vergleichsbasierten Suchalgorithmus. Wir nehmen an, dass der Algorithmus die Suche durch Vergleiche ausführt, d.h., Elemente miteinander vergleicht und aufgrund des Resultats Ja/Nein-Entscheidungen trifft.

Einen solchen Algorithmus können wir als *Entscheidungsbaum* darstellen. Jeder Knoten in diesem Baum entspricht einem Vergleich, und die beiden Kindknoten entsprechen der Entscheidung, die der Algorithmus aufgrund des Vergleichs fällt (Ja/Nein). Die Blätter des Entscheidungsbaums entsprechen den Rückgabewerten des Algorithmus. Ist eine Suche nach einem Element b erfolgreich, dann kann b an *jeder* der n Positionen des Arrays stehen. Für jede dieser n möglichen Ergebnisse der Suche muss der Entscheidungsbaum mindestens einen Knoten enthalten. Zusätzlich muss es einen Knoten für "nicht gefunden" geben, also muss die Gesamtanzahl der Knoten mindestens $n+1$ betragen.

Die Anzahl von Vergleichen, die ein Algorithmus im schlechtesten Fall ausführt, entspricht exakt der *Höhe* des Baums. Diese ist definiert als die maximale Anzahl von Knoten auf einem Weg von der *Wurzel* (dem "obersten" Knoten) zu einem *Blatt* (einem Knoten ohne Nachfolger). Wir beobachten, dass ein Baum der Höhe h höchstens

$$2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} = 2^h - 1 < 2^h \tag{2}$$

viele Knoten hat. Daraus folgt

$$n + 1 \leq \text{Anzahl Knoten im Entscheidungsbaum} < 2^h \Rightarrow h > \log_2(n+1). \tag{3}$$

Da die Anzahl der im schlimmsten Fall ausgeführten Vergleiche h beträgt und unsere Argumentation für *beliebige* Entscheidungsbäume gilt, haben wir also bewiesen dass jedes vergleichsbasierte Suchverfahren auf einem sortierten Array im schlechtesten Fall $\Omega(\log n)$ viele Vergleiche durchführt.

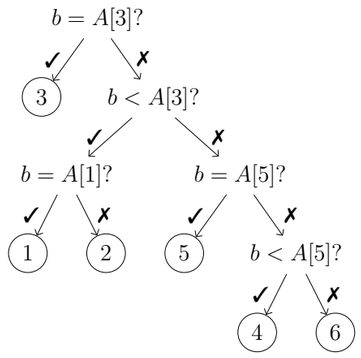


Abb. 1.1 Binäre Suche auf einem Array der Länge 6 als Entscheidungsbaum visualisiert. Da sich das zu suchende Element im Erfolgsfall auf sechs möglichen Positionen befinden kann, hat der Entscheidungsbaum mindestens diese Anzahl von Knoten. Im ersten Schritt wird $A[3]$ mit b verglichen. Bei Gleichheit ist der Algorithmus fertig, ansonsten werden $A[1]$ bzw. $A[5]$ verglichen, usw. Da der Baum Höhe 4 hat (also aus fünf Leveln inklusive Blättern besteht), ist die binäre Suche auf einem Array mit sechs Elementen nach spätestens vier Vergleichen fertig.

1.2.1 Untere Schranke

Untere Schranke

Um zu beweisen, dass die Suche in sortierten Arrays im schlimmsten Fall immer $\Omega(\log n)$ viele Vergleiche ausführt, betrachten wir wieder einen *beliebigen* vergleichsbasierten Suchalgorithmus. Wir nehmen an, dass der Algorithmus die Suche durch Vergleiche ausführt, d.h., Werte miteinander vergleicht und aufgrund des Resultats (Ja/Nein) Entscheidungen trifft.

Einen solchen Algorithmus können wir als *Entscheidungsbaum* darstellen. Jeder innere Knoten in diesem Baum entspricht einem Vergleich, und die beiden Kindknoten entsprechen der Fortsetzung, die der Algorithmus je nach Ausgang des Vergleichs fällt (Ja/Nein). Da jeder Knoten höchstens zwei Kinder hat, sprechen wir von einem *binären* Baum. Die Knoten in der untersten Ebene, auch *Blätter* genannt, entsprechen den Rückgabewerten des Algorithmus. Ist eine Suche nach einem Element b erfolgreich, dann kann b an jeder der n Positionen des Arrays stehen. Für jede dieser n möglichen Ergebnisse der Suche muss der Entscheidungsbaum mindestens ein Blatt enthalten. Zusätzlich muss es einen Knoten für „nicht gefunden“ geben, also muss die Gesamtanzahl der Blätter mindestens $n+1$ betragen. Die Anzahl Knoten ist somit auch mindestens $n+1$, denn jedes Blatt ist ja insbesondere ein Knoten.[3]

Die Anzahl von Vergleichen, die ein Algorithmus im schlechtesten Fall ausführt, entspricht exakt der *Höhe* des Baums.[4] Diese ist definiert als die maximale Anzahl von Kanten auf einem Weg von der *Wurzel* (dem "obersten" Knoten) zu einem *Blatt* (einem Knoten ohne Nachfolger). Wir beobachten, dass ein binärer Baum der Höhe

[3] Das ist natürlich sehr grob abgeschätzt. Als Übung können Sie auch versuchen, sich direkt zu überlegen, wie viele Blätter ein Baum der Höhe h haben kann. Das führt zu der minimal besseren unteren Schranke $h \geq \log_2(n+1)$.

[4] Statt der Höhe spricht man manchmal auch von der Tiefe des Baumes oder eines Knotens. Die Wurzel ist also in Tiefe 0, das tiefste Blatt in Tiefe h.

h höchstens

$$2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1 < 2^{h+1} \tag{2}$$

viele Knoten hat. Daraus folgt

$$n + 1 \leq \text{Anzahl Knoten im Entscheidungsbaum} < 2^{h+1}, \tag{3}$$

was aufgelöst $h > \log_2(n+1) - 1 = \Omega(\log n)$ ergibt. Da die Anzahl der im schlimmsten Fall ausgeführten Vergleiche h beträgt und unsere Argumentation für *beliebige* Entscheidungsbäume gilt, haben wir also bewiesen, dass jedes vergleichsbasierte Suchverfahren auf einem sortierten Array im schlechtesten Fall $\Omega(\log n)$ viele Vergleiche durchführt.

Ein Argument von dieser Form nennt man auch *informationstheoretisches Argument*. Es formalisiert die Intuition, dass ein vergleichsbasierter Algorithmus durch h Vergleiche nur "h Bits an Information" sammeln kann und dadurch nur höchstens 2^h viele Zustände voneinander unterscheiden kann. Eine Verallgemeinerung dieses Prinzips führt auf das Konzept der *Entropie*, das in der Informatik an vielen verschiedenen Stellen auftritt.

1.2.2 Einsortierung

Einsortierung

Oft will man mit binärer Suche nicht nur herausfinden, ob b in dem Array vorkommt. Manchmal will man (auch) wissen, an welche Position wir b im Array einfügen müssten, damit das Array dabei sortiert bleibt.[5] Das kann man durch leichte Modifikation von binärer Suche ebenfalls erreichen. Falls b im Array vorkommt, geben wir ja sowieso die Position von b zurück. Falls b nicht im Array vorkommt, so kann man sich davon überzeugen, dass bei Terminierung $\ell = r + 1$ gilt[6], und dass $A[r] < b < A[\ell]$ ist. Die gesuchte Position ist in diesem Fall also ℓ.

[5] Wenn das Element b mehrfach im Array auftritt, gibt es mehrere mögliche Antworten.

[6] Da man die while-Schleife verlassen hat, muss $\ell > r$ sein. Man muss sich also überlegen, dass wir ℓ niemals auf einen grösseren Wert als $r+1$ setzen, und umgekehrt r niemals auf einen kleineren Wert als $\ell - 1$ setzen.