Suchen auf Arrays

Dozent: Prof. Dr. Michael Eichberg

Kontakt: michael.eichberg@dhbw.de, Raum 149B

Version: 1.1.4

Quelle: Die Folien sind teilweise inspiriert von oder basierend auf Lehrmaterial von Prof. Dr. Ritterbusch

---

Folien: https://delors.github.io/theo-algo-suchen_auf_arrays/folien.de.rst.html
https://delors.github.io/theo-algo-suchen_auf_arrays/folien.de.rst.html.pdf

Fehler melden: https://github.com/Delors/delors.github.io/issues

1. Einführung

Skalierung von Daten

Welche Skalierung haben gesuchte Daten sind im Array?

nominal:	Nur Vergleich auf Gleichheit, keine natürliche Ordnung oder Zahlbegriff.
ordinal:	Es gibt Größenvergleiche und damit eine Sortierung, aber kein Zahlbegriff.
kardinal:	Es gibt Größenvergleiche und Zahlbegriff.

- *Unsortiert oder nominal* führt (zunächst) zur linearen Suche.
- *Ordinale und kardinale Werte* können sortiert werden für binäre Suche.
- *Kardinale Größen* können modelliert werden für interpolierende Suche.

Hinweis

Für unsere Betrachtung gehen wir im Folgenden davon aus, dass die Daten sortiert sind. Beim Vergleich der Algorithmen beschränken wir uns auf eine Betrachtung der Anzahl der Elementzugriffe.

- Ein Beispiel für eine *nominal* skalierte Datenmenge wäre die Menge der Farben. Es gibt keine natürliche Ordnung der Farben, und es gibt auch keinen natürlichen Zahlenbegriff, der die Farben beschreibt. Ein weiteres Beispiel ist eine Liste von Wohnorten.4
- Ein Beispiel für eine *ordinale* skalierte Datenmenge wäre die Menge der Kleidergrößen (S,M,L,XL,...). Es gibt eine natürliche Ordnung der Kleidergrößen, aber es gibt keinen natürlichen Zahlenbegriff, der die Kleidergrößen beschreibt. Ein weiteres Beispiel ist die Bewertung von Filmen auf einer Skala von 1 bis 5 Sternen.

Lineare Suche

```
Algorithm linearSearch(A,n,needle)
    for i= 1,...,n do
        if A[i] == needle then
            return i
    return nil
```

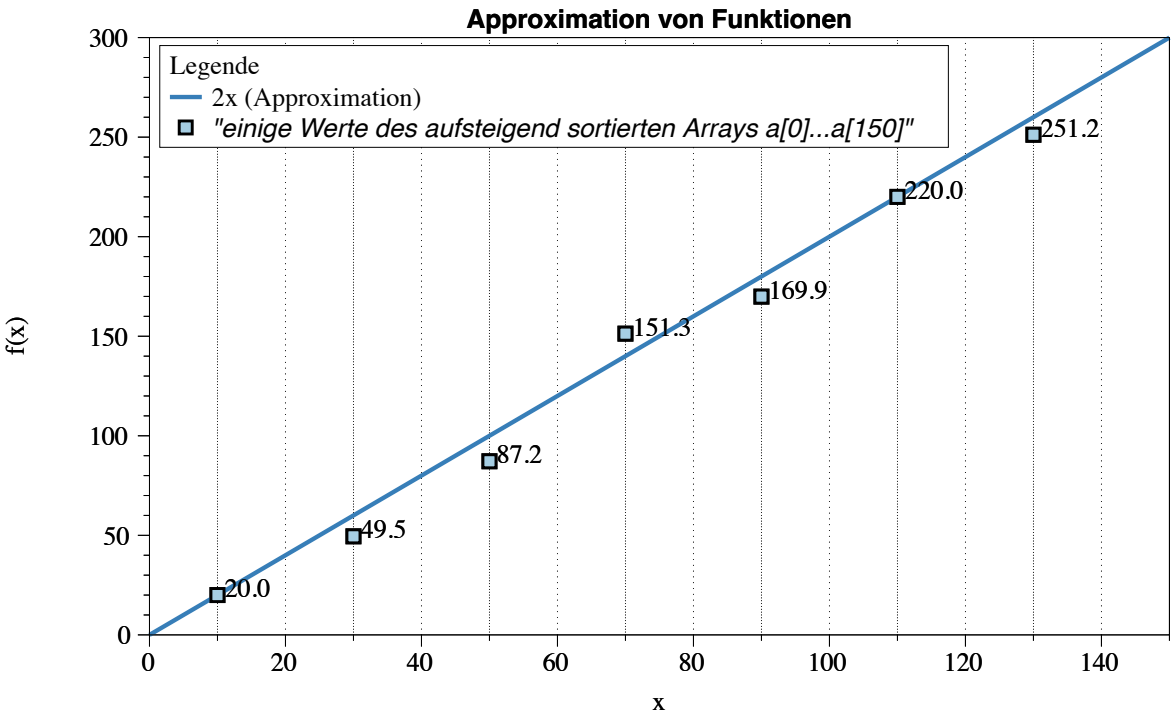
Laufzeit und Elementzugriffe kann asymptotisch durch $O(n)$ abgeschätzt werden.

Binäre Suche

```
1  Algorithm binarySearch(A,l,u,needle)
2      upper = u
3      lower = l
4      repeat
5          pos = round((upper+lower)/2)
6          value = A[pos]
7          if value == needle then
8              return pos
9          else if value > needle then
10             upper = pos-1
11         else
12             lower = pos + 1
13     until upper < lower
14     return nil
```

Laufzeit ist $O(\log(n))$, genauer im Schnitt $\log_2(n) - 1$ Zugriffe.

Effizientere Suche bei linearer Verteilung



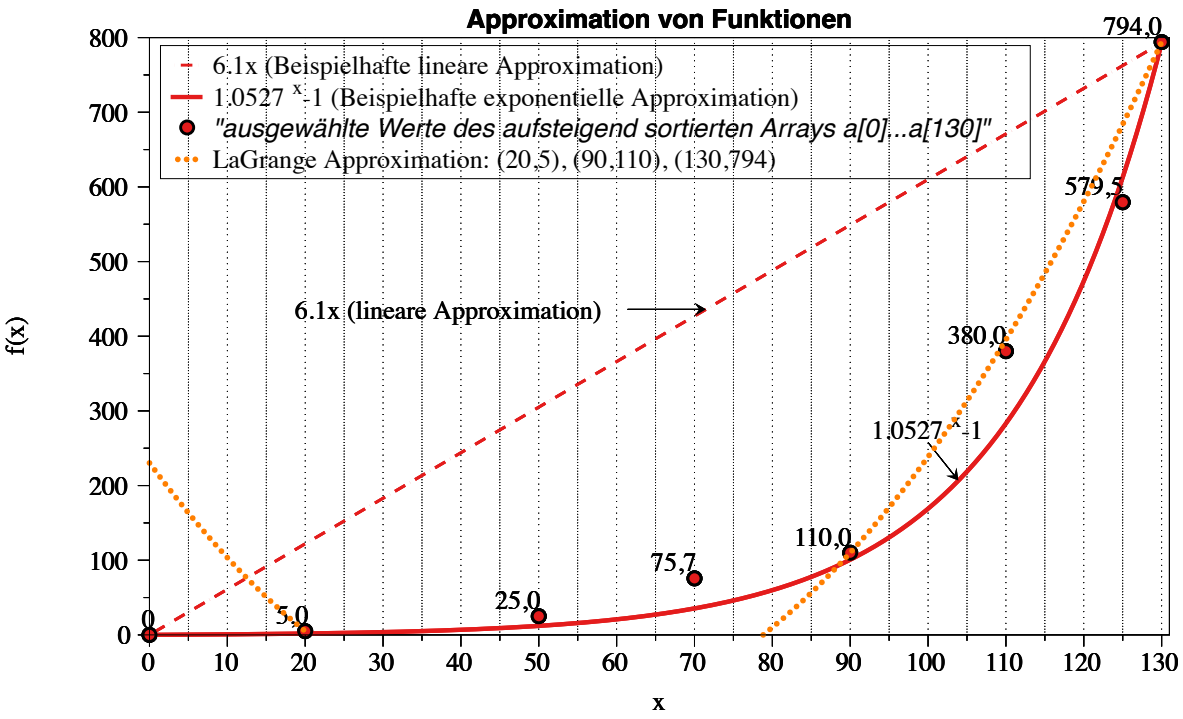
In diesem Beispiel gehen wir davon aus, dass die Werte *im Wesentlichen* linear verteilt sind. Das bedeutet, dass die Differenz zwischen zwei aufeinanderfolgenden Werten immer gleich ist.

Sei beispielsweise ein Array `a` mit folgenden Werten geben (Auszug):

Index i	Wert
i = 10	a[i = 10] = 20.0
...	...
i = 30	49.5
...	...
i = 50	87.2
...	...
i = 70	151.3
...	...
i = 90	169.9
...	...
i = 110	220.0
...	...
i = 130	251.2

Wenn man jetzt exemplarisch die Paare: $(i = 10, a[i] = 20.0)$, und $(i = 110, a[i] = 220)$ betrachtet, dann kann man zu dem Schluss kommen, dass die Funktion $f(x) = 2.0 \cdot x$ eine Approximation der Verteilung der Werte ist. Würde man also nach dem Wert $a[i] = y = 170$ suchen wollen, dann wäre es gut als erstes den Wert von `a[85]` zu überprüfen, $170 = 2 \cdot x \rightarrow \frac{170}{2} = 85 = i$.

Effizientere Suche bei exponentieller Verteilung



In diesem Beispiel gehen wir davon aus, dass die Werte *im Wesentlichen* exponentiell verteilt sind. Das bedeutet, dass die Differenz zwischen zwei aufeinanderfolgenden Werten immer größer wird.

Sei beispielsweise ein Array `a` mit folgenden Werten geben (Auszug):

index i	a[i]
0	0
...	...
20	5
...	...
50	25
...	...
70	75.7
...	...
90	110
...	...
110	380
...	...
125	579.5
...	...
130	794

Wenn man jetzt exemplarisch die Paare: $(i = 20, a[i] = 5.0)$, und $(i = 130, a[i] = 794)$ betrachtet, und eine lineare Approximation durchführt, dann könnte man zu dem Schluss kommen, dass die Funktion $f(x) = 6.1 \cdot x$ eine gute Approximation ist.

Würde man eine quadratische Approximation mit Hilfe von Lagrange durchführen,

zum Beispiel mit den Werten $(i = 20, a[i] = 5.0)$, $(i = 90, a[i] = 110)$, und $(i = 130, a[i] = 794)$. Dann wäre der Fehler zwischen der realen Verteilung und der angenommen deutlich geringer, da die quadratische Funktion die Werte besser approximiert.

In diesem Fall Fall wäre die Funktion: $p(x) = \frac{39}{275}x^2 - \frac{141}{10}x + \frac{2533}{11}$ In diesem Fall können wir die Position des Wertes 650 im Array besser abschätzen (durch die Aufstellung der Umkehrfunktion und dann einsetzen von 650): ≈ 123.

Warnung

Eine vernünftige Interpolation ist nur dann möglich, wenn die Verteilung der Werte im Wesentlichen bekannt ist.

Approximation der Verteilung

!! Wichtig

Wenn wir die Verteilung der Werte kennen, können wir effizientere Algorithmen entwickeln.

Beispiel

Wenn wir wissen, dass die Werte quadratisch verteilt sind (`Array[10] a = { 1, 4, 9, 16, ..., 100 }`), und wir zum Beispiel wissen, dass der kleinste Wert im Array 1 und der größte Wert 100 (an Stelle/mit Index 10) ist, den wir im Array gespeichert haben, dann macht es „keinen“ Sinn den Wert 85 oder 5 in der Mitte zu suchen! (85 findet sich vermutlich an Stelle $9 = \lfloor \sqrt{85} \rfloor$).

Interpolation mit Lagrange-Polynomen

Speichert unser Array kardinal skalierte Daten, so können diese modelliert werden.
Das einfachste Prinzip ist die Polynominterpolation mittels Lagrange-Polynomen.

Das Ziel ist es, ein Polynom $p(x)$ zu finden, das eine Funktion $f(x)$ an einer gegebenen Menge von Punkten $(x_1, y_1), \ldots, (x_n, y_n)$ *exakt* interpoliert. Das heißt:

$$p(x_i) = y_i \quad \text{für alle } i = 1, \ldots, n$$

Satz

Das Lagrange-Interpolationspolynom $p(x)$ wird als Summe von Lagrange-Basispolynomen $l_i(x)$ aufgebaut:

$$p(x) = \sum_{i=1}^{n} (y_i \cdot l_i(x))$$

wobei $l_i(x)$, das i-te Lagrange-Basispolynom, gegeben ist durch:

$$l_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}$$

Sind n Tupel $(x_n, y_n) \in \mathbb{R}^2$ reeller Zahlen gegeben mit $x_l \neq x_m$ für $l \neq m$.

Das Lagrange-Interpolationspolynom hat dann höchstens den Grad $n - 1$ und es gilt $p(x_i) = y_i$ für alle $i = 1, \ldots, n$.

Beispiel

Gegeben sein die zwei Punkte: $(x_1, y_1) = (1, 2)$ und $(x_2, y_2) = (3, 4)$.

Das Lagrange-Polynom $p(x)$ wäre dann:

1. $$l_1(x) = \frac{x - x_2}{x_1 - x_2} = \frac{x - 3}{1 - 3} = \frac{3 - x}{2}$$
2. $$l_2(x) = \frac{x - x_1}{x_2 - x_1} = \frac{x - 1}{3 - 1} = \frac{x - 1}{2}$$
3. $$p(x) = y_1 \cdot l_1(x) + y_2 \cdot l_2(x) = 2 \cdot \frac{3 - x}{2} + 4 \cdot \frac{x - 1}{2} = x + 1$$

Nach Ausmultiplizieren und Zusammenfassen ergibt das ein Polynom, das durch beide Punkte verläuft.

Wenn zwei Punkte gegeben sind, ist das Lagrange Polynom somit:

$$p(x) = y_1 \cdot \frac{x - x_2}{x_1 - x_2} + y_2 \cdot \frac{x - x_1}{x_2 - x_1}$$

Bei drei Punkten ist das Lagrange Polynom somit:

$$
\begin{aligned}
p(x) = \quad & y_1 \cdot \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} + \\
& y_2 \cdot \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} + \\
& y_3 \cdot \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)}
\end{aligned}
$$

Der Grad unseres Lagrange-Polynoms ist immer um 1 kleiner als die Anzahl der gegebenen Punkte (die Terme des Basispolynom sind nur für $j \neq i$ definiert). Das bedeutet, dass wir für zwei Punkte ein lineares Polynom erhalten, für drei Punkte ein quadratisches Polynom, für vier Punkte ein kubisches Polynom, und so weiter. Weiterhin stellt die Konstruktion sicher, dass wir durch alle gegebenen Punkte gehen.

Übung

1.1. Bestimme p(2)

Bestimmen Sie direkt $p(2)$ für das quadratische Polynom mit den Eigenschaften:

$$p(-10) = 3, p(-8) = 1, p(-4) = -1$$

1.2. Bestimme p(-1)

Für die gegebenen Punkte, bestimmen Sie erst das Lagrange Polynom $p(x)$ im Allgemeinen und rechnen Sie dann den Wert für $p(-1)$ aus.

$$p(2) = 4, p(4) = 6, p(7) = 3$$

Interpolierende Suche - lineare Approximation

Beispiel

Gegeben:
Vom Array `a` sei bekannt: `a[1] = 0`, `a[20] = 30` und `a[40] = 120`.

Frage: Ist der Wert `50` im Array enthalten?

Lösung: Das Lagrangepolynom $p(x)$ mit $p(30) = 20$ und $p(120) = 40$ lautet:

$$p(x) = 20 \cdot \frac{x - 120}{30 - 120} + 40 \cdot \frac{x - 30}{120 - 30}$$

Für den gesuchten Wert `50` ergibt sich als zu untersuchende Position:

$$p(50) = 20 \cdot \frac{50 - 120}{30 - 120} + 40 \cdot \frac{50 - 30}{120 - 30} = \frac{220}{9} \approx 24$$

Bemerkung

Wir möchten die Position des Wertes 50 im Array abschätzen! Deswegen sind im linearen Modell die Paare $(x_1, y1) = (30, 20)$ und $(x_2, y_2) = (120, 40)$ zu wählen. D. h. die Indizes sind unsere y-Werte.

Eine binäre Suche würde in diesem Fall mit der Position $\frac{40+20}{2} = 30$ beginnen.

Hinweis

Das Lagrangepolynom kann per Konstruktion die Position der Werte 30 und 120 perfekt bestimmen:

$$p(30) = 20 \cdot \frac{30 - 120}{30 - 120} + 40 \cdot \frac{30 - 30}{120 - 30} = 20$$

$$p(120) = 20 \cdot \frac{120 - 120}{30 - 120} + 40 \cdot \frac{120 - 30}{120 - 30} = 40$$

Interpolierende Suche - quadratische Approxi.

Beispiel

Gegeben:

Vom Array `a` sei bekannt: `a[1] = 0`, `a[20] = 30` und `a[40] = 120`.

Frage: Ist der Wert `50` im Array enthalten?

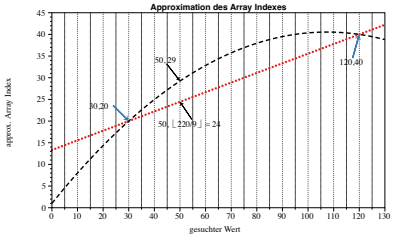
Lösung: $p(x)$ mit $p(0) = 1$, $p(30) = 20$ und $p(120) = 40$ lautet:

$$\begin{aligned} p(x) = \quad & 1 \cdot \frac{(x-30)(x-120)}{(0-30)(0-120))} + \\ & 20 \cdot \frac{(x-0)(x-120)}{(30-0)(30-120)} + \\ & 40 \cdot \frac{(x-0)(x-30)}{(120-0)(120-30)} \end{aligned}$$

Für den gesuchten Wert `50` ergibt sich als zu untersuchende Position:

$$p(50) \approx 29$$

Interpolierende Suche - Vergleich



- Auf gleichverteilten Daten hat die lineare Interpolationssuche O(log log n).
- Auf anderen Verteilungen ist lineare Interpolation oft schlechter als binäre Suche.
- Quadratische Interpolation hat ein erweitertes Modell und schlägt binäre Suche häufig.

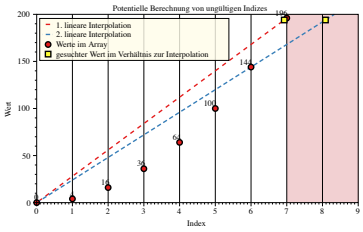
Lineare interpolierende Suche

```
1  Algorithm linearInterpolatingSearch(A,needle)
2      lower := 1                // index auf das kleinste Element
3      upper := length(A)        // index auf das größte Element
4      vL := A[lower]
5      if vL == needle then return lower
6      vU := A[upper]
7      if vU == needle then return upper
8      while upper > lower do
9          pos := round(lower·(needle−vU)/(vL−vU) +
10                      upper·(needle−vL)/(vU−vL))
11         pos := max(lower + 1, min(upper - 1, pos))
12         value := A[pos]
13         if value == needle then return pos
14         else if value < needle then
15             lower := max(pos, lower+1), vL = A[lower]
16         else
17             upper := min(pos, upper-1), vU = A[upper]
18     return nil
```

Die Korrektur von `pos` in Zeile 11 (`pos := max(lower + 1, min(upper - 1, pos))`) stellt sicher, dass `pos` immer strikt zwischen `lower` und `upper` liegt. Dies ist insbesondere deswegen notwendig, weil die Interpolation nicht immer exakt ist. Stellen Sie sich zum Beispiel vor, dass die Daten polynomiell skaliert sind und sie (in Unkenntnis der echten Verteilung) die lineare Interpolationssuche verwenden. In diesem Fall kann es zu folgender Situation kommen:

Die Werte im Array seien: [0, 4, 16, 36, 64, 100, 144, 196] (zu Grunde liegt die Funktion $4x^2$) und Sie suchen nach dem Wert 194.

Im ersten Schritt würde die lineare Interpolationssuche den Wert 194 auf Position 7 schätzen, was nutzlos wäre, aber erst einmal kein Problem verursachen würde. Da der Wert 194 aber nicht im Array enthalten ist, würde die Suche den Wert für die obere Grenze um eins korrigieren. Jetzt würde die lineare Interpolation aber mit den Werten des Arrays an Stelle 0 und 6 erfolgen (A[0] = 0 und A[6] = 144). Das Ergebnis wäre die 2. Funktion (blau) und der Wert 194 würde auf Position 8 geschätzt, was außerhalb des Arrays liegt.



Folgen die Werte im Array einer logarithmisch Verteilung, dann würde die umgekehrte Situation eintreten, d. h. es könnte am unteren Ende des Arrays zu einem ähnlichen Problem kommen, da dann die Werte oberhalb der geraden liegen würden.

Wenn der berechnete Index außerhalb des Bereichs ist, dann kann der Algorithmus auch einfach `nil` zurückgeben, da der Wert dann nicht im Array enthalten ist.

Exponentielle Suche im sortierten (unbeschränkten) *Array*

```
1 Algorithm ExponentialSearch(A,needle)
2     i = 1
3     while A[i] < needle do
4         i = i * 2
5     return BinarySearch(A,floor(i/2) + 1,i,needle)
```

Die Idee ist erst mit einer exponentiellen Schrittweite zu springen, um dann mit einer binären Suche den Wert zu finden. Die Laufzeit ist $O(\log(i))$ wobei i die Position des gesuchten Wertes ist. Die Laufzeit ist also $O(\log(n))$.

Übung

1.3. Wer sucht, der findet 5?

Folgende Werte sind vom Array A bekannt:

$$A[1] = -27,\ A[15] = 13,\ A[29] = 29$$

Gesucht wird der potentielle Index des Wertes `5`. Welcher Index `i` sollte als nächstes untersucht werden bei binärer, linearer oder quadratisch interpolierender Suche?

1.4. Wer sucht, der findet -1?

Folgende Werte sind vom Array A bekannt:

$$A[1] = -13,\ A[7] = -4,\ A[13] = 11$$

Gesucht wird der potentielle Index des Wertes `-1`. Welcher Index `i` sollte als nächstes untersucht werden bei binärer, linearer oder quadratisch interpolierender Suche?

Übung

1.5. Lineare Interpolierende Suche

Setzen Sie den Algorithmus für die lineare interpolierende Suche in einer Programmiersprache Ihrer Wahl um.

Testen Sie den Algorithmus mit folgenden Arrays:

```
A = [1, 3, 5, 7, 9, 11, 13, 15] # linear verteilt (2x-1)

B = [0, 7, 13, 22, 27, 32, 44, 49] # approx. linear verteilt (approx. 7x)

C = [0, 2, 16, 54, 128, 250, 432, 686] # quadratisch verteilt (4x^2)
```

Wie viele Schritte (im Sinne von Schleifendurchläufen) sind maximal notwendig, um festzustellen ob ein Wert im Array enthalten ist oder nicht?

Übung

1.6. Exponentiell Interpolierende Suche

Implementieren Sie den Algorithmus für die exponentiell interpolierende Suche in einer Programmiersprache Ihrer Wahl. (Ggf. müssen Sie noch die passende binäre Suche implementieren).

Gegeben sei die folgende Funktion, die als Generator fungiert. (x sei eine natürliche Zahl).

$$f(x) = \text{round}\left(\frac{1}{1+e^{-x}} \cdot 100\,000\,000\right)$$

Testen Sie ob 99999996 ein Wert der Funktion ist und geben Sie den Index (x) zurück.

Wann macht es Sinn die exponentiell interpolierende Suche zu verwenden?

2. Selbstanordnende Arrays

Suchen auf Arrays mit spezieller Ordnung

- Sind die Daten nominal skaliert, oder sagt die Ordnung der Werte im Array nichts über die Zugriffshäufigkeit aus, so können Arrays auf Basis der Zugriffe sortiert werden.
- Erfordert prinzipiell eine lineare Suche, die es gilt soweit möglich zu beschleunigen.
- Anwendung(-sgebiete):
  - Cache-Zugriffe, Verwaltung von virtuellem Speicher
  - Wenn Werte häufiger verlangt werden als andere, so besitzen die Anfragen eine Wahrscheinlichkeitsverteilung.
  - Die Verteilung wird durch Abzählen angenähert, da sie nicht bekannt ist. Darauf basierend werden die Werte entsprechend sortiert.

Strategien zur Anordnung

Definition

Ein Array A ist gemäß **frequency count** oder **FC-Regel** sortiert, wenn für alle Werte gilt, dass $c(A[k]) \geq c(A[j])$ wenn $k < j$ und $c(x)$ die realisierte Häufigkeit des Wertes x darstellt.

Hinweis

Es wird typischerweise lokal getauscht, um die Ordnung herzustellen.

Definition

Ein Array A ist gemäß **move to front** oder nach der **MF-Regel** sortiert, wenn bei Auftritt eines Wertes $A[k]$ in der Folge mit der ersten Position $A[1]$ oder $A[0]$ vertauscht wird, sollte der Wert noch nicht an der ersten Stelle stehen.

Definition

Ein Array A ist gemäß **transpose** oder nach der **T-Regel** sortiert, wenn bei Auftritt eines Wertes $A[k]$ in der Folge mit der Position davor $A[k-1]$ vertauscht wird, sollte der Wert noch nicht an der ersten Stelle stehen.

Strategien zur Anordnung - Diskussion

- Die FC-Regel erfordert das Mitführen der Häufigkeit der Werte. Die MF-Regel und die T-Regel sind einfacher zu implementieren, da sie nur die Reihenfolge der Werte im Array verändern.
- Für MF-Regel und T-Regel gibt es worst-case Aufrufsequenzen, die immer zu den schlechtesten Laufzeiten führen.
- Die MF-Regel nimmt eher starke Änderungen vor und reagiert schnell.
- Die T-Regel nimmt eher schwache Änderungen vor und ist stabiler.

Zusammenfassung

Die Bewertung sollte an Hand der tatsächlichen Daten erfolgen:

- Liegen Häufigkeitsinformationen vor, so ist die FC-Regel sinnvoll.
- Die MF-Regel ist für sich ändernde Verteilungen sinnvoller, die T-Regel für stabilere Situationen.

Übung

2.1. A = [1,2,3,4,5] selbstanordnend sortieren

Das Array `A = [1,2,3,4,5]` soll selbstanordnend sortiert werden. Die gesuchten Werte sind: `1,2,3,2,3,2,1,5`. Bestimmen Sie die Anordnung des Arrays nach jedem Zugriff für die Sortierungen nach MF-Regel, T-Regel und FC-Regel. Füllen Sie die nachfolgende Tabelle aus:

x	MF-Regel	T-Regel	FC-Regel	Häufigkeiten pro Wert
1				
2				
3				
2				
3				
2				
1				
5				

Übung

2.2. A = [1,2,3,4,5,6] selbstanordnend sortieren

Das Array `A = [1,2,3,4,5,6]` soll selbstanordnend sortiert werden. Danach werden die folgenden Werte in der angegebenen Reihenfolge gesucht: `5,1,6,2,3,6,5`. Bestimmen Sie die Anordnung des Arrays nach jedem Zugriff für die Sortierungen nach MF-Regel, T-Regel und FC-Regel. Füllen Sie die nachfolgende Tabelle aus:

x	MF-Regel	T-Regel	FC-Regel	Häufigkeiten
5				
1				
6				
2				
3				
6				
5				

3. Textsuche

Arrays und Textsuche

Texte können als unsortierte Arrays von Zeichen verstanden werden. Eine typische Frage ist hier das Finden von Textsequenzen im Text.

Einfache Textsuche

```
1  Algorithmus NaiveTextSearch(text,needle)
2      n = length(text)
3      m = length(needle)
4      for i = 1,...,n-m + 1 do
5          j = 0
6          while text[i + j] == needle[j + 1] do
7              j = j + 1
8              if j == m then
9                  return i // Found at i
10     return nil
```

Bemerkung

Die Laufzeit der einfachen Textsuche kann asymptotisch durch $O(n \cdot m)$ abgeschätzt werden.

Beispiel bei einfacher Suche nach `aaab` in `aaaaaaaab`:

```
a a a a a a a a b
-----------------

a a a b
  a a a b
    a a a b
      a a a b
        a a a b
          a a a b
```

Sind so viele Vergleiche notwendig?

Knuth-Morris-Pratt Verfahren - Grundlagen

Das Verfahren von Knuth-Morris-Pratt vermeidet unnötige Vergleiche, da es zunächst die Suchwortteile auf den größten Rand, also das größte Prefix, das auch Postfix ist, untersucht.

Definition: Präfix, Postfix und Rand

Für ein Wort $w = (w_1, \ldots, w_n)$ sind die Präfixe $p^{(k)} = (w_1, \ldots, w_k)$ und die Postfixe $q^{(k)} = (w_{n-k+1}, \ldots, w_n)$ für $0 \leq k \leq n$.

Ist $p^{(k)} = q^{(k)} = r^{(k)}$ für ein $0 \leq k < n$, so ist $r(k)$ ein Rand von w.

Für $k < n$ werden $p^{(k)}$ und $q^{(k)}$ auch echte Prä- und Postfixe genannt.

Beispiel/Idee

Text	010110101
Gesucht/Muster	010101
Übereinstimmung	✓✓✓✓✗

Beobachtungen:

1. Wir haben an Stelle 5 ein Mismatch.
2. Wenn wir im Text das Muster um eine Stelle nach rechts verschoben suchen, so haben wir garantiert wieder ein Mismatch.

? Frage

Wie weit kann man also das Muster im Allgemeinen verschoben werden ohne ein Vorkommen zu übersehen?

Beispiel/Idee

	1.	2.
Text	01101100	0102111
Gesucht/Muster	01100	010201
Übereinstimmungen	✓✓✓✓✗	✓✓✓✓✗

Beobachtungen bzgl.:

1. Beim Mismatch an Stelle 5 kann das Muster „nur“ um 3 Stellen nach rechts verschoben werden.
2. Beim Mismatch an Stelle 5 kann das Muster um 4 Stellen nach rechts verschoben werden.

Wie weit wir das Muster verschieben können, hängt also vom Rand des Teils des Musters ab, der bereits übereinstimmt.

Beispiel

Das Wort $aufkauf$ hat die *echten* Präfixe und Postfixe:

$$\{p^{(k)} : 0 \leq k < n\} = \{\varepsilon, a, au, auf, aufk, aufka, aufkau\}$$

$$\{q^{(k)} : 0 \leq k < n\} = \{\varepsilon, f, uf, auf, kauf, fkauf, ufkauf\}$$

und die Ränder:

$$\{r^{(k)} : 0 \leq k < n\} = \{\varepsilon, auf\}.$$

Das bedeutet, dass wenn $aufkauf$ erkannt wurde, die letzten drei Buchstaben schon den nächsten Treffer einleiten können, wie beispielsweise in $aufkaufkauf$.

Das KMP-Verfahren fängt nicht immer von vorne an, sondern prüft, ob ein Rand eines Präfixes $-\varepsilon$ ausgenutzt werden kann. Dazu werden die entsprechenden größten Ränder bestimmt.

Beispiel: ananas

Präfixe $setminus\varepsilon$	Größter Rand	Länge des Randes
a	ε	0
an	ε	0
a̲nā	a	1
a̲n̲āñ	an	2
a̲n̲a̲ñā	ana	3
ananas	ε	0

Beispiel: axaaxax

Präfixe $setminus\varepsilon$	Größter Rand	Länge des Randes
a	ε	0
ax	ε	0
a̲xā	a	1
a̲xaā	a	1
a̲x̲aāx̄	ax	2
a̲x̲a̲āx̄ā	axa	3
a̲x̲aaxāx̄	ax	2

Die Idee ist also, dass wir beim Musterabgleich nach einem Mismatch, wenn der übereinstimmende Teil einen Rand hat, beim Abgleich des Musters an einer späteren Stelle - basierend auf der Größe des Randes - weitermachen können. Wir müssen also nicht immer das ganze Muster von vorne anfangen zu vergleichen.

Übung

3.1. Ränder und Randlängen bestimmen

Bestimmen Sie die Ränder und die Längen der $Präfixe - \varepsilon$ für die Worte:

1. $tultatul$
2. $eikleike$
3. $okokorok$
4. $trattrad$

Knuth-Morris-Pratt Verfahren

```
Algorithm ComputePrefixFunction(needle)
    m = length(needle)
    sei B[1...m] ein Array // Array für die Längen der Ränder der Teilworte
    B[1] = 0
    j = 0 // j ist die Länge des Randess
    for i = 2,...,m do
        j = j + 1
        while j > 0 and needle[j] ≠ needle[i] do
            if j > 1 then
                j = B[j-1] + 1
            else
                j = 0
        B[i] = j
    return B
```

Komplexität: $O(m)$

```
Algorithm KMP(text,needle)
    n = length(text), m = length(needle)
    B = ComputePrefixFunction(needle)
    q = 0               // Anzahl der übereinstimmenden Zeichen
    R = []              // Ergebnisliste der Indizes der Übereinstimmungen
    for i = 1,...,n do
        while q > 0 and needle[q + 1] ≠ text[i] do
            q = B[q]    // ... die nächsten Zeichen stimmen nicht überein
        if needle[q + 1] == text[i] then
            q = q + 1   // Übereinstimmung
        if q == m then
            R append (i - m + 1)
            q = B[q]    // Suche nach nächster Übereinstimmung
    return R
```

Komplexität: $O(n + m)$

Details ComputePrefixFunction

Die Funktion $ComputePrefixFunction$ berechnet die größten Werte der Präfixe für das Suchwort $needle$ der Länge m und gibt diese als Array (B) zurück. Das Array B enthält somit die größten Ränder der Präfixe $needle[1, \ldots, i]$. (Der Wert von $B[1]$ ist immer 0, da es keinen Rand gibt.)

Beispiel für eine KMP-Textsuche

Gesucht wird `ananas` **in** `saansanananas`

```
   s a a n s a n a n a n a s
i  ──────────────────────────
1  a
...
3    a n
...
5      a n a
...
11           a n a n a s       Beim Auftreten des Mismatch (ln 7) ist
...                            q=5 und wird auf p[5]=3 (ln 8) gesetzt
13               a n a n a s
```

Bemerkung

Dargestellt sind die Fälle, in denen ein Mismatch auftritt. `i` ist der Index des aktuellen Zeichen im Text, das mit dem Muster verglichen wird.

Übung

3.2. KMP-Algorithmus

Bestimmen Sie die Randlängen der Muster und stellen Sie die Teilschritte bei der Durchführung des KMP-Algorithmus zur Suche des Wortes/Muster im Text dar.

Stellen Sie insbesondere die Fälle dar in denen ein Mismatch auftritt.

Muster	Text
`aaab`	`aaaaaaaab`
`barbara`	`abbabarabarbarbara`

Boyer-Moore-Algorithmus (vereinfacht)

Beobachtung

Häufig ist das Alphabet des Textes größer als das des Musters.

- Der Algorithmus vergleicht das Muster (Pattern) von rechts nach links mit dem Text.

  Viele andere Algorithmen führen die Vergleiche von links nach rechts durch.
- Der Boyer-Moore-Algorithmus nutzt dies aus, indem er die Verschiebung des Musters anhand des letzten Zeichens des Musters und des Textes vornimmt.

Wird beispielsweise das Wort `Banane` im Text `Orangen, Ananas und Bananen` gesucht, so wird zunächst die Sprungtabelle für das verwendete Alphabet in Bezug auf das Suchwort (`Banane` mit Länge 6) bestimmt:

Zeichen im Text	␣	,	A	B	O	a	d	e	g	n	r	s	u
Sprung	6	6	6	5	6	2	6	0	6	1	6	6	6

```
O r a n g e n , ␣ A n a n a s ␣ u n d ␣ B a n a n e n
B a n a n e
          B a n a n e
            B a n a n e
              B a n a n e
                B a n a n e
                            B a n a n e
                              B a n a n e
                                B a n a n e
                                  B a n a n e
```

Bemerkung

Unterschrichen sind die durchgeführten Vergleiche. Die Verschiebung des Musters erfolgt anhand des letzten Zeichens des Musters und des Textes, dass nicht übereinstimmt. Dabei ist die Verschiebung durch das Zeichen des Textes gegeben, das nicht mit dem Muster übereinstimmt.

Komplexität

Im guten und häufigen Fall erreicht das Verfahren $O(\frac{n}{m})$, aber in speziellen Fällen ist auch $O(n \cdot m)$ möglich.

Bei der Sprungtabelle handelt es sich um eine Tabelle, die *für jedes Zeichen des Alphabets des Textes* die Verschiebung des Musters angibt, wenn das Zeichen im Text mit dem Muster nicht übereinstimmt. Die Zeichen `A`, `O`, `d`, `g`, `r`, `s`, `u`, `,` und das Leerzeichen haben die größte Verschiebung, da sie nicht im Muster vorkommen. Das Zeichen `e` hat die kleinste Verschiebung, da es das letzte Zeichen des Musters ist. Das Zeichen `n` hat eine Verschiebung von 1, da es im Muster als vorletztes Zeichen vorkommt, das Zeichen `a` hat eine Verschiebung von 2, da das späteste Vorkommen an drittletzer Stelle ist, und das Zeichen `B` hat eine Verschiebung von 5, da es nur einmal vorkommt und das erste Zeichen des Musters ist.

3.3. „belli“

Suchen Sie das Wort

```
belli
```

im Text

```
It is a dark time for the Rebellion.
```

3.4. "barbara"

Suchen Sie das Wort

```
barbara
```

im Text

```
abbabarabarbarbara
```

4. Suche nach dem n-ten Element

Suche nach dem n-ten Element - Einführung

- Ist das Array sortiert, so ist die Suche nach dem n-ten Element trivial und hat eine Laufzeit von $O(1)$.
- Ist das Array nicht sortiert, so ist die Suche nach dem n-ten Element nicht trivial.

  Wir unterscheiden:

  1. wird das Array (im Folgenden) auch noch sortiert gebraucht, so ist es am effizientesten dieses erst zu sortieren, um dann das n-te Element auszulesen. Die Laufzeit beträgt dann - mit der Wahl eines geeigneten Sortierverfahrens - $O(n \log n)$.
  2. Ist eine Sortierung nicht erforderlich/gewünscht, so können wir mit Hilfe von Teile-und-Herrsche-Verfahren das n-te Element auch effizienter bestimmen.

Suche nach dem n-ten Element mittels Quickselect

```
1  Algorithm Quickselect(A,k)     // gesucht ist das k größte Element
2      if length(A) == 1 then return A[0]
3      pivot := A[length(A)-1]    // ein bel. Element als Pivot (hier das letzte)
4      lows := []                 // Elemente kleiner als Pivot
5      highs := []                // Elemente größer als Pivot
6      pivotsCount := 0           // Anzahl der Pivot-Elemente
7      for x in A do              // Partitionierung ...
8          if x < pivot then lows.append(x)
9          else if x > pivot then highs.append(x)
10         else pivotsCount := pivotsCount + 1
11
12     if k < length(lows) then
13         return Quickselect(lows, k)
14     else if k < length(lows) + pivotsCount then
15         return pivot           // das k-te Element ist ein Pivot-Element
16     else
17         return Quickselect(highs, k - len(lows) -  pivotsCount)
```

Hinweis

In einer realen Implementierung sollte das Pivot-Element zufällig gewählt werden, um - für den Fall, dass das Array sortiert ist - die Laufzeit zu verbessern.

Hinweis

Der Quickselect Algorithmus kann auch *in-place* implementiert werden, d. h. ohne zusätzlichen Speicherbedarf. Dies setzt voraus, das die ursprüngliche Reihenfolge der Elemente nicht erhalten bleiben muss.

Beispielanwendung: Bestimmung des Medians

```
Algorithm FindMedian(A) // A ist _nicht sortiert_
    n = length(A)
    if n % 2 == 1 then // d. h. wir haben eine ungerade Anzahl von Elementen in A
        return Quickselect(A, floor(n / 2))
    else // gerade Anzahl von Elementen in A
        left = Quickselect(A, floor(n / 2) - 1)
        right = Quickselect(A, floor(n / 2))
        return (left + right) / 2
```

Übung

4.1. n-te Element bestimmen

1. Bestimmen Sie den Median für das Array `A = [23, 335, 2, 24, 566, 3, 233, 54, 42, 6, 667, 7, 5, 7, 7]`. Wenden Sie dazu den Algorithmus `FindMedian` (inkl. `Quickselect-Algorithmus`) an.
2. Geben Sie weiterhin nach jeder Partitionierung im Quickselect Algorithmus den aktuellen Zustand an (d. h. nach Zeile 11 in Quickselect).

Array A	k	Lows	Pivot	Pivots Count	Highs
[...]	<K>	[...]	<P>	<#P>	[...]

4.2. Komplexität von Quickselect

Bestimmen Sie die Komplexität des Quickselect-Algorithmus im schlechtesten Fall, im Durchschnittsfall und im besten Fall.

Komplexitätsanalyse

Zur Bestimmung der Komplexität kann man entweder das Master Theorem anwenden oder die Anzahl der Schritte für die Partitionierung bestimmen und die Summe der Schritte aufstellen.

Geometrische Reihen

Die Summenformel für eine geometrische Reihe ($S_n = a + ar + ar^2 + \cdots + ar^{n-1}$) lautet:

$$S_n = a \cdot \frac{1 - r^n}{1 - r} \quad \text{für} \quad r \neq 1$$

Mit:

S_n:	Summe der ersten n Glieder der geometrischen Reihe.
a:	Das erste Glied der Reihe.
r:	Der Quotient (Verhältnis aufeinanderfolgender Glieder).
n:	Die Anzahl der Glieder.

Für n gegen unendlich und $|r| < 1$ gilt somit:

$$S = \frac{a}{1 - r} \quad \text{für} \quad |r| < 1$$