ETH zürich

Departement Informatik

Informatikgrundlagen

Binäre Suche, Max. Teillisten

Hans-Joachim Böckenhauer
Dennis Komm

Herbst 2021 – 9. Dezember 2021

Daten suchen

Lineare Suche

Lineare Suche

Laufe durch Liste von links nach rechts und vergleiche jedes Element mit dem gesuchten

- Einfachste Möglichkeit der Suche
- Funktioniert auch in ungeordneten Daten
- Braucht bis zu n Vergleiche auf Liste der Länge n, falls gesuchtes Element an letzter Stelle ist (oder nicht vorkommt)
- Komplexität in $\mathcal{O}(n)$

Lineare Suche

```
def linsearch(data, searched):
    index = 0
    while index < len(data):
        if data[index] == searched:
            return index
        index += 1
    return -1
```

```
def linsearch(data, searched):
    if searched in data:
        return True
    else:
        return False
```

```
def linsearch(data, searched):
    return searched in data
```

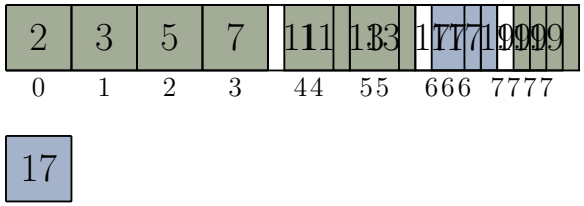
Daten suchen

Binäre Suche

Binäre Suche

Beispiel

Gegeben eine Liste der ersten 8 Primzahlen, finde heraus, an welcher Position Primzahl 17 ist



Binäre Suche

Nutze aus, dass Daten sortiert sind

- Zwei Variablen `left` and `right`
- Diese geben **Suchraum** an
- Betrachte Wert in der Mitte (Index `current`)
- Ist dieser der gesuchte, sind wir fertig
- Ist dieser zu klein, ist auch alles links von ihm zu klein

⇨ `left = current + 1`

- Ist dieser zu gross, ist auch alles rechts von ihm zu gross

⇨ `right = current - 1`

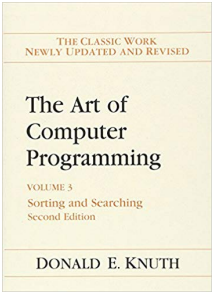
Binäre Suche

Daten **suchen** und sortieren sind zwei der grundlegendsten Aufgaben von Informatikerinnen und Informatikern

Die erste binäre Suche wurde 1946 veröffentlicht (und das Prinzip war bereits lange davor bekannt), allerdings ist die erste für alle n korrekt funktionierende Version erst 14 Jahre später erschienen

«Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky. . . »

–Donald Knuth



Aufgabe – Binäre Suche

Implementieren Sie die binäre Suche

- als Python-Funktion
- mit «Zeigern» `left`, `right` und `current`
- Am Anfang ist `left = 0` und `right = len(data) - 1`
- Verkleinern Sie den Suchraum in jedem Schritt wie beschrieben
- Wenn Element gefunden wird, wird seine Position zurückgegeben
- Sonst wird -1 zurückgegeben

Binäre Suche

```
def binsearch(data, searched):
    left = 0
    right = len(data) - 1
    while left <= right:
        current = (left + right) // 2
        if data[current] == searched:
            return current
        elif data[current] > searched:
            right = current - 1
        else:
            left = current + 1
    return -1
```

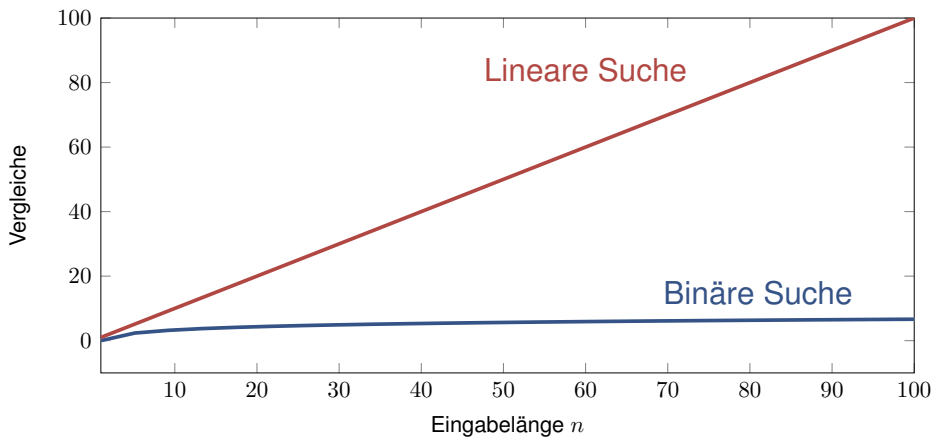

Suchen

Komplexität binärer Suche

Komplexität binärer Suche

- Am Anfang hat Liste n Elemente
- Mit jeder Iteration wird der **Suchraum** auf die Hälfte reduziert
- Nach der ersten Iteration $n/2$ Elemente
- Nach der zweiten Iteration $n/4$ Elemente
- ...
- Nach wie vielen Iterationen x ist nur noch ein Element übrig?
- $n/2^x = 1 \iff n = 2^x \iff x = \log_2 n$
- Komplexität in $\mathcal{O}(\log_2 n)$

Komplexität binärer Suche



Komplexität binärer Suche

Laufzeit der binären Suche im schlechtesten Fall darstellen

- Wir verwenden wieder eine Variable `counter`, um Vergleiche zu zählen
- Algorithmus wird auf sortierten Listen mit den Werten 1 bis n laufen gelassen
- Der Wert von n wächst mit jeder Iteration um 1
- Am Anfang ist n dabei 1, am Ende $1\,000\,000$
- Gesucht wird immer das erste Element 1
- Ergebnis wird in einer Liste gespeichert und mit `matplotlib` geplottet

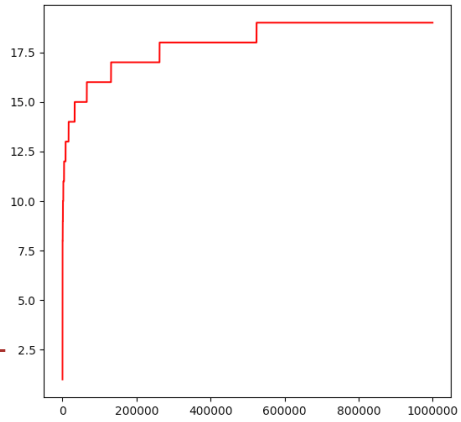
Komplexität binärer Suche

Worst Case

```
values = []
data = [1]


for i in range(1, 1000001):
    data.append(data[-1] + 1)
    values.append(binsearch(data, 1))


plt.plot(values, color="red")
plt.show()
```



Komplexität binärer Suche

Was, wenn Daten unsortiert sind?

- Lineare Suche funktioniert auch in unsortierten Listen und ist in $\mathcal{O}(n)$
- Bei mehrfacher Suche kann einmaliges Sortieren sich lohnen
- Sortieren ist in $\mathcal{O}(n \log_2 n)$ und damit langsamer als lineare Suche
- Binäre Suche ist in $\mathcal{O}(\log_2 n)$ und damit allerdings wiederum viel schneller als lineare Suche

Wann lohnt sich Sortieren?
Wenn öfter als $\log_2 n$ Mal gesucht werden muss

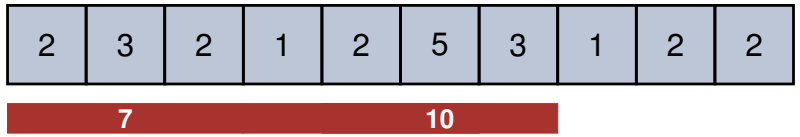
Daten suchen und durchsuchen

Teillisten

Maximale Teilliste fester Länge

Bekannt aus Projekt 2

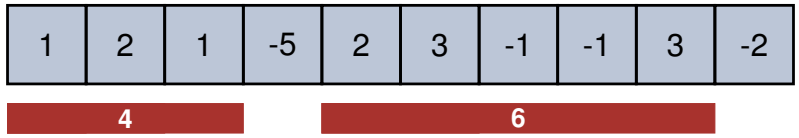
- Gegeben n positive ganze Zahlen
- Finde Teilliste der Länge k maximaler Summe



Maximale Teilliste

Betrachten wir ein verwandtes Problem

- Gegeben n ganze Zahlen (positiv oder negativ)
- Finde Teillliste mit maximaler Summe



Maximale Teilliste

Kadanes Algorithmus

- Jede Teilliste (auch eine maximale) hat eine Endposition x
- Betrachte alle Möglichkeiten
- Für jedes $x \geq 1$
  - Betrachte die maximale Teilliste A_{x-1} mit Endposition $x-1$
  - **Entweder** ist diese Teil der maximalen Teilliste A_x mit Endposition x
  - **Oder** nicht (dann ist A_x Teilliste der Länge 1)

$$\text{Maximum}(A_x) = \max\{\text{Maximum}(A_{x-1}) + \text{Wert von } x, \text{Wert von } x\}$$

Maximale Teilliste

Ist maximale Teilliste A_{x-1} enthalten?

1	2	1	-5	2	3	-1	-1	3	-2

-1 4

4 6

Aufgabe – Maximale Teilliste

Implementieren Sie Kadanes Algorithmus

- als Python-Funktion
- Speichere aktuelles Maximum
- Teste, ob die maximale Teilliste, die eine Position früher endet, Teil davon sein sollte
- Geben Sie nur den maximalen Wert zurück

Das Maximum von zwei Zahlen kann mit der Funktion `max()` berechnet werden

Aufgabe – Maximale Teilliste

```
def find_sub(daten):
    max_hier = daten[0]
    maximum = max_hier

    for i in range(1, len(daten)):
        max_hier = max(max_hier + daten[i], daten[i])
        maximum = max(maximum, max_hier)
    return maximum
```