Algorithmen und Datenstrukturen 1

2. Vorlesung

Martin Middendorf und Peter F. Stadler

Universität Leipzig
Institut für Informatik
middendorf@informatik.uni-leipzig.de, studla@bioinf.uni-leipzig.de

Einfache Suchverfahren

- Lineare Listen
- Sequentielle Suche
- Binäre Suche
- Weitere Suchverfahren auf sortierten Feldern
  - Sprungsuche
  - Exponentielle Suche
  - Interpolationssuche
- Auswahlproblem

Lineare Listen

- Endliche Folge von Elementen eines Grundtyps (Elementtyps)

  $<a_1, a_2, \ldots a_n>$ falls $n > 0$

  leere Liste $< >$ falls $n=0$

- Position von Elementen in der Liste ist wesentlich

Wesentliche **Implementierungsalternativen**

- **Sequentielle Speicherung** (**Reihung, Array**). Eigenschaften:
  1. statische Datenstruktur (hoher Speicheraufwand)
  2. wahlfreie Zugriffsmöglichkeit über (berechenbaren) Index
  3. **1-dimensionale** vs. **mehrdimensionale** Arrays
- **Verkettete Speicherung**. Eigenschaften:
  1. dynamische Datenstruktur (Einfügen und Löschen einfach)
  2. nur sequentielle Navigation
  3. Varianten: **einfache** vs. **doppelte** Verkettung etc.

Sequenzielle Suche

Suche nach Element mit Schlüsselwert x:

falls unbekannt, ob die Elemente der Liste nach ihren Schlüsselwerten sortiert sind, ist die Liste sequentiell zu durchlaufen und elementweise zu überprüfen (**sequenzielle Suche**)

Kosten

- erfolglose Suche erfordert n Schleifendurchläufe
- erfolgreiche Suche verlangt im ungünstigsten Fall n Schlüsselvergleiche (und n-1 Schleifendurchläufe)
- mittlere Anzahl von Schlüsselvergleichen bei erfolgreicher Suche:

$$C_{avg}(n) = \frac{1}{n} \cdot \sum_{i=1}^{n} i = \frac{n+1}{2}$$

Binäre Suche

Auf sortierten Listen können Suchvorgänge effizienter durchgeführt werden

1. Versuch: **Naiver Ansatz**: Sequentielle Suche auf sortierten Listen
   → bringt nur geringe Verbesserungen für erfolglose Suche

2. Versuch: **Binärsuche**
   Suche nach Schlüssel x in Liste mit aufsteigend sortierten Schlüsseln:

```
Falls Liste leer ist, endet die Suche erfolglos.
Sonst: Betrachte Element L[m] an mittlerer Position m.
Falls x = L[m] dann ist das gesuchte Element gefunden.
Falls x < L[m], dann durchsuche die linke Teilliste von Position 1
  bis m-1 nach demselben Verfahren
Sonst (x > L[m]) durchsuche die rechte Teilliste von Position m+1
  bis Ende nach demselben Verfahren
```

   Kosten (Anzahl der Schlüsselvergleiche) für erfolgreiche Binärsuche:

$$C_{min}(n) = 1 \qquad C_{max}(n) = \lceil \log_2(n+1) \rceil \qquad C_{avg}(n) = \log_2(n+1) - 1, \text{ für große } n$$

   → Verbesserung von O(n) auf O(log n)

Die Funktionen floor und ceiling

- Wichtige Funktionen (in Mathematica oder C usw.) Definition in C:
  - double floor(double x) { Größte ganze Zahl nicht größer als x }

    Oder: floor(x) ist der ganzzahlige Anteil von x.

    Beispiel: floor(3.14) = 3

    Andere Notation: **floor**(x) = $\lfloor x \rfloor$

    Eigenschaften:

    1. $\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$.
    2. $\lfloor k+x \rfloor = k + \lfloor x \rfloor$ (für bel. ganze Zahl k)
  - double ceil(double x) { Kleinste ganze Zahl nicht kleiner als x }

    Beispiele: ceiling(2.3) = 3, ceiling(2) = 2 and ceiling(−2.3) = −2

    Andere Notation:

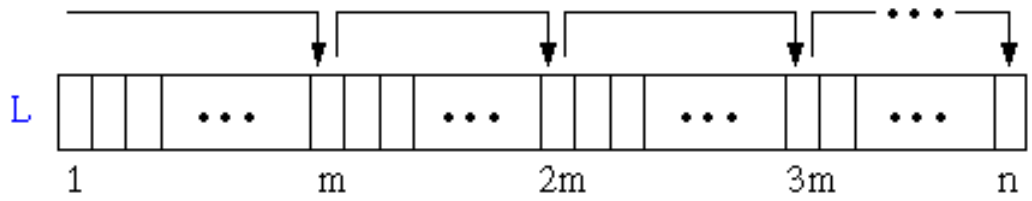
    ceiling(x) = $\lceil x \rceil$

    Eigenschaften:

    1. $x \leq \lceil x \rceil < x+1$
    2. $\lfloor k \rfloor = \lceil k \rceil$ (für bel. ganze Zahl k)
    3. $\lfloor k/2 \rfloor + \lceil k/2 \rceil = k$ (für bel. ganze Zahl k)

Sprungsuche

Prinzip: der sortierte Datenbestand wird zunächst in Sprüngen überquert, um Abschnitt zu lokalisieren, der ggf. den gesuchten Schlüssel enthält.



Einfache Sprungsuche:

- konstante Sprünge zu Positionen m, 2m, 3m, ...
- Sobald x <= L[i] mit i=j·m (j=1,2,...) wird im Abschnitt L[(j-1)m+1] bis L[j·m] sequentiell nach dem Suchschlüssel x gesucht.

Mittlere Suchkosten: ein Sprung koste a; ein sequenzieller Vergleich b Einheiten

$$C_{avg}(n) = \frac{1}{2}a \cdot \frac{n}{m} + \frac{1}{2}b(m-1)$$

Optimale Sprungweite: $m = \sqrt{(a/b)n}$

- falls a=b => $m = \sqrt{n}$, $c_{avg}(n) = a\sqrt{n} - a/2$

Komplexität: $O(\sqrt{n})$

Sprungsuche - Details

Mittlere Suchkosten :

Zur Erinnnerung : Bei k Elementen sind die mittleren Suchkosten durch $C_{avg} = \frac{k+1}{2}$ gegeben.

Hier : Max. Zahl der Sprünge ist $\lfloor n/m \rfloor$ aber der letzte Sprung kann gespart werden. Mittlere Sprungkosten also

$$\frac{1}{2} a \left\lfloor \frac{n}{m} \right\rfloor$$

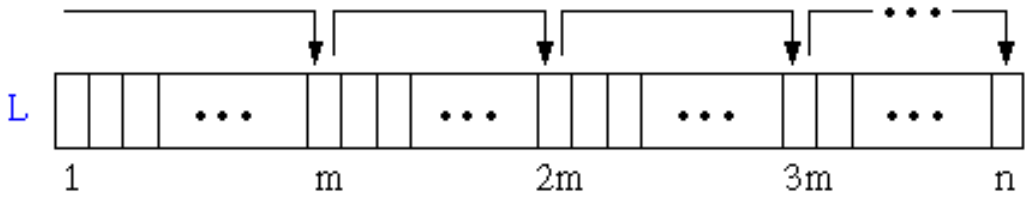
Die Suche im Intervall muss nur über die Positionen $(j\text{-}1)m+1 \ldots jm-1$ erfolgen, da die Position jm schon die nächste Einsprungstelle ist. Das sind m -1 Positionen. Die Position jm - 2 ist aber die letzte die abgefragt werden muss, wenn man sicher weiß dass der Schlüssel in der Liste gespeichert ist.

Bleiben also m - 2 Schlüsselvergleiche und der Beitrag zu C_{avg} ist somit

$$\frac{1}{2} b(m-1)$$

Optimale Sprungweite : Betrachte

$$\frac{\partial}{\partial m} C_{avg} = \frac{1}{2} \frac{\partial}{\partial m} \left(a \frac{n}{m} + b(m-1) \right) = 0$$



Zwei-Ebenen-Sprungsuche

Prinzip: Statt sequentieller Suche im lokalisierten Abschnitt wird wiederum eine Quadratwurzel-Sprungsuche angewendet, bevor dann sequenziell gesucht wird

Mittlere Suchkosten: $C_{avg}(n) \leq \frac{1}{2} \cdot a \cdot \sqrt{n} + \frac{1}{2} \cdot b \cdot n^{\frac{1}{4}} + \frac{1}{2} \cdot c \cdot n^{\frac{1}{4}}$

a Kosten eines Sprungs auf der ersten Ebene

b Kosten eines Sprungs auf der zweiten Ebene

c Kosten für einen sequentiellen Vergleich

Verbesserung durch optimale Abstimmung der Sprungweiten m_1 und m_2 der beiden Ebenen

- Mit a = b = c ergeben sich als optimale Sprungweiten $m_1 = n^{\frac{2}{3}}$ und $m_2 = n^{\frac{1}{3}}$

und mittlere Suchkosten: $C_{avg}(n) = \frac{3}{2} \cdot a \cdot n^{\frac{1}{3}}$

Verallgemeinerung zu **n-Ebenen-Verfahren** ergibt ähnlich günstige Kosten wie Binärsuche

Anwendung: Sprungsuche vorteilhaft, wenn Binärsuche nicht anwendbar ist
(z.B. bei blockweisem Einlesen der sortierten Sätze vom Externspeicher)

Exponentielle Suche

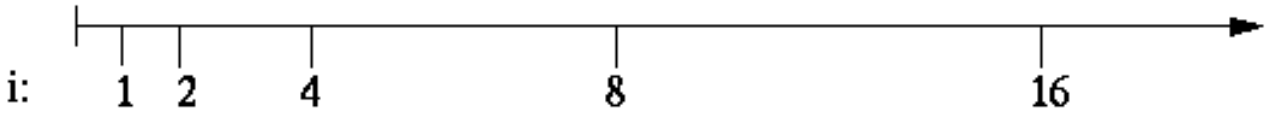
Anwendung: wenn Länge des Suchbereichs n zunächst unbekannt bzw. sehr groß

Vorgehensweise:

- für Suchschlüssel x wird zunächst obere Grenze für den zu durchsuchenden Abschnitt bestimmt

```
int i = 1; while (x > L[i]) i=2*i;
```

- für i > 1 gilt für den auf diese Weise bestimmten Suchabschnitt: `L[i/2] < x <= L[i]`
- suche innerhalb des Abschnitts mit irgendeinem Verfahren



Analyse:

Annahme: Die sortierte Liste enthält nur positive, ganzzahlige Schlüssel ohne Duplikate

=> Schlüsselwerte wachsen mindestens so stark wie die Indizes i der Elemente

=> i wird höchstens $\log_2 x$ mal verdoppelt, denn die Zahl der Sprünge ist gleich $\log_2 i$

- Bestimmung des gesuchten Intervalls erfordert maximal $\log_2 x$ Schlüsselvergleiche
- Suche innerhalb des Abschnitts (z.B. mit Binärsuche) erfordert auch höchstens $\log_2 x$ Schlüsselvergleiche

Gesamtaufwand $O(\log_2 x)$

Interpolationssuche

Idee: Schnellere Lokalisierung des Suchbereichs in dem Schlüsselwerte selbst betrachtet werden, um "Abstand" zum Suchschlüssel x abzuschätzen

Vorgehensweise: Nächste Suchposition pos wird aus den Werten ug und og der Unter- und Obergrenze des aktuellen Suchbereichs wie folgt berechnet

$$pos = ug + \frac{x - L[ug]}{L[og] - L[ug]} \cdot (og - ug)$$

Analyse:

Sinnvoll, wenn Schlüsselwerte im betreffenden Bereich einigermaßen gleichverteilt

- erfordert dann im Mittel lediglich $\log_2 \log_2 n + 1$ Schlüsselvergleiche

Im schlechtesten Fall (stark ungleichmäßige Werteverteilung) entsteht jedoch linearer Suchaufwand (O(n))

Häufigkeitsgeordnete lineare Listen

Prinzip: Ordne die Liste nach Zugriffhäufigkeiten

Sinnvoll, falls die Zugriffshäufigkeiten für die einzelnen Elemente einer Liste sehr unterschiedlich sind

Analyse: Mittlere Suchkosten: $c_{avg}(n) = 1 \cdot p_1 + 2 \cdot p_2 + 3 \cdot p_3 + ... + n \cdot p_n = \Sigma\, i \cdot p_i$

für Zugriffswahrscheinlichkeiten p_i und bei sequenzieller Suche

→ zur Minimierung der Suchkosten sollte Liste direkt so aufgebaut oder umorganisiert werden, daß $p_1 \geq p_2 \geq ... \geq p_n$

Beispiel: Zugriffsverteilung nach 80-20-Regel

- 80% der Suchanfragen betreffen 20% des Datenbestandes und
- von diesen 80% wiederum 80% (also insgesamt 64%) der Suchanfragen richten sich an 20% von 20% (insgesamt 4%) der Daten

Erwarteter Suchaufwand $c_{avg}(n) = ???$

Problem: Da Zugriffshäufigkeiten meist vorab nicht bekannt sind, werden **selbstorganisierende** (adaptive) Listen benötigt.

Selbstorganisierende Listen 1

Anwendung: Nützlich, falls häufig gleiche Objekte gesucht werden (Beispiel: Suchbegriffe bei Suchmaschinen)

Prinzip:

- gesucht wird immer sequenziell von vorn
- die Reihenfolge der Objekte in der Liste passt sich dem Suchverhalten an (dazu gibt es verschiedene Regeln)

Ansatz 1: FC-Regel (Frequency count)

- führen einen Häufigkeitszähler pro Element
- jeder Zugriff auf ein Element erhöht dessen Häufigkeitszähler um 1
- falls erforderlich, wird danach die Liste lokal neu geordnet, so dass die Häufigkeitszähler der Elemente eine absteigende Reihenfolge bilden
- hoher Wartungsaufwand und Speicherplatzbedarf

Selbstorganisierende Listen 2

Ansatz 2: T-Regel (Transpose)

- das Zielelement eines Suchvorgangs wird dabei mit dem unmittelbar vorangehenden Element vertauscht
- häufig referenzierte Elemente wandern (langsam) an den Listenanfang

Ansatz 3: MF-Regel (Move-to-Front)

- das Zielelement eines Suchvorgangs wird nach jedem Zugriff an die erste Position der Liste gesetzt
- relative Reihenfolge der übrigen Elemente bleibt gleich
- in jüngster Vergangenheit referenzierte Elemente sind am Anfang der Liste (Lokalität kann gut genutzt werden)

Auswahlproblem

Aufgabe: Finde das i-kleinste Element in einer Liste L mit n Elementen

- Spezialfälle: kleinstes (Minimum), zweitkleinstes, … Element, mittlerer Wert (Median), größtes (Maximum)

Bei sortierter Liste: trivial

Bei unsortierter Liste: Minimum/Maximum-Bestimmung hat lineare Kosten O(n)

Algorithmus: Einfache Lösung für i-kleinstes Element

```
j := 0
solange j < i:
     Bestimme kleinstes Element in L und entferne es aus L ;
     erhöhe j um 1
gib Minimum der Restliste als Ergebnis zurück
```

Komplexität (für nichttriviales i): $O(n^2)$

JEDOCH: es geht aber besser

Teile und Herrsche (Divide-and-Conquer)

Allgemeines Verfahren, um große Probleme schneller zu lösen:

1. Zerlege das Problem in zwei oder mehrere Teilprobleme.
2. Löse jedes Teilproblem einzeln, und zwar rekursiv nach der gleichen Methode.
3. Konstruiere die Gesamtlösung aus den Teillösungen.

Dieses Verfahren wird in Schritt 2 rekursiv angewendet, bis hinreichend triviale (kleine) Probleme vorliegen, die sich mit anderen Mitteln lösen lassen.

Frage: Wieso lässt sich damit die Komplexität verringern?

Sehen wir uns zunächst Beispiele an

Teile und Herrsche 1: Suche in sortierter Liste Sequenzielle Suche -> Binärsuche

Teilungsschritt: Die Liste wird in der Mitte geteilt

Triviale Liste: Liste der Länge 1, wird mit sequenzieller Suche durchsucht

Gesamtlösung: Erfolgreich, falls in einem Teil gefunden

Unser Algorithmus zur Binärsuche lässt sich umformen zu:

```
Falls die Liste die Länge 1 hat: Prüfe, ob es sich um das gesuchte
    Element handelt und gib JA oder NEIN zurück.
Sonst zerlege die Liste in der Mitte und prüfe durch Vergleich des
    gesuchten Elements mit der Trennstelle, in welcher Hälfte das
    gesuchte Element liegen muss.
Wende diesen Algorithmus rekursiv auf diese Teilliste an.
```

Zur Erinnerung: Verbesserung der Zeitkomplexität von O(n) zu O(log n)

Teile und Herrsche 2: Berechnung der maximalen Teilsumme

Idee: Wir können wieder die gegebene Folge in zwei Teile zerlegen und die maximalen Teilsummen für beide Teile bestimmen.

ABER: Die maximale Teilfolge könnte gerade die Schnittstelle einschließen

DESHALB: Zusätzliche Größen: Rechtes und linkes Randmaximum einer Folge F

- **rechte Randfolge** von F = Teilfolge von F, die bis zum rechten Rand (Ende) von F reicht
- **rechtes Randmaximum** von F: maximale Summe aller rechten Randfolgen
- analog: **linke Randfolge**, **linkes Randmaximum**

Tag	1	2	3	4	5	6	7	8	9	10
Gewinn/Verlust (Folge)	+5	-6	+4	+2	-5	+7	-2	-7	+3	+5

Teile und Herrsche 2: Berechnung der maximalen Teilsumme

Rekursiver (Divide-and-Conquer-) Algorithmus für maximale Teilsumme

- falls Eingabefolge F nur aus einer Zahl z besteht, nimm Max(z,0)
- falls F wenigstens 2 Elemente umfasst:
  - zerlege F in etwa zwei gleich große Hälften links und rechts
  - bestimme maximale Teilsumme, ml, sowie rechtes Randmaximum, rR, von links
  - bestimme maximale Teilsumme, mr, sowie linkes Randmaximum, lR, von rechts
  - das Maximum der drei Zahlen ml, rR+lR, und mr ist die maximale Teilsumme von F

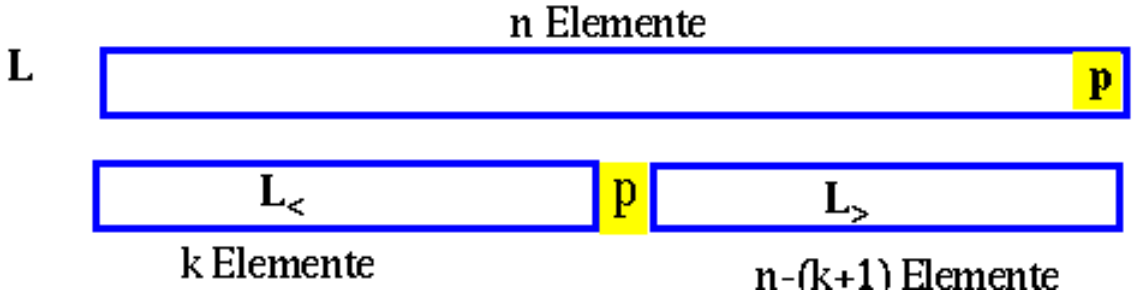
Komplexitätsverbesserung?

Tag	1	2	3	4	5	6	7	8	9	10
Gewinn/Verlust (Folge)	+5	-6	+4	+2	-5	+7	-2	-7	+3	+5

Teile und Herrsche 3: Auswahlproblem

Suche i-kleinstes Element von n paarweise unterschiedlichen Elementen

1. bestimme Pivot-Element p
2. Teile die n Elemente bezüglich p in 2 Gruppen: Gruppe 1 enthält die k Elemente <p; Gruppe 2 die n-k-1 Elemente >p sind
3. falls i=k+1, dann ist p das Ergebnis.
   falls i <= k wende das Verfahren rekursiv auf Gruppe 1 an;
   falls i > k+1: verwende Verfahren rekursiv zur Bestimmung des i-(k+1)-te Element in Gruppe 2



Komplexität: In Schritt 2 wird die ganze verbleibende Liste durchlaufen. Im ersten Durchlauf n, im zweiten ca. n/2, dann ca. n/4 usw., insgesamt ca. 2n, also O(n)

→ Verbesserung von $O(n^2)$ zu O(n)

Teile und Herrsche 4

Weitere Beispiele folgen beim Thema Sortieren, z.B. *Quicksort*.

Frage: In welchem Sinne war unsere schnelle Multiplikation eine Anwendung von Teile-und-Herrsche?

Komplexitätsreduktion bei Teile und Herrsche

Annahme: wir wollen ein Problem der Größe $n = 2^r$ lösen

Beobachtungen:

- die maximale Rekursionstiefe ist $r = \log_2 n$
- die Anzahl der Teilprobleme in Rekursionstiefe k ist 2^k
- muss in jeder Tiefe nur ein Teilproblem gelöst werden (wie z.B. beim Suchen), so sind es insgesamt $r = \log_2 n$ Teilprobleme
- muss jedes Teilproblem gelöst werden, so sind n elementare Probleme zu lösen und diese Lösungen zu verknüpfen. Das macht speziell Sinn für Probleme der Komplexität $O(n^2)$ oder schlechter (z.B. maximale Teilsumme)
- der Faktor log n bei der Komplexität entsteht auf natürliche Weise durch Teile-und-Herrsche

Aufgabe: Erklären Sie damit, dass Teile-und-Herrsche keine Verbesserung bei sequenzieller Suche in einer ungeordneten Liste bringt

Zusammenfassung: Suchen

Sequentielle Suche

- Default-Ansatz zur Suche
- lineare Kosten O(n)

Binärsuche

- setzt Sortierung voraus
- Teile-und-Herrsche-Strategie
- wesentlich schneller als sequentielle Suche
- Komplexität: O(log n)

Weitere Suchverfahren auf sortierten Arrays für Sonderfälle

- Sprungsuche
- exponentielle Suche
- Interpolationssuche

Auswahlproblem

- auf unsortierter Eingabe mit linearen Kosten lösbar