TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa / Arnd Poetzsch-Heffter

AG Softech
FB Informatik
TU Kaiserslautern

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Klassische Sortier- und Suchalgorithmen

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Motivation

- Suchen und Sortieren sind Teilprobleme, die in einer Vielzahl von Programmen auftreten
- Beispiel: Musiksammlungen, Profile in sozialen Netzwerken, Berechnung von Median, Erstellen von Histogrammen
- Große Datenmengen erfordern effiziente Algorithmen (Laufzeit und Speicherbedarf)

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Thematischer Überblick

- Binäre Suche
- Sortierverfahren auf Arrays (Sortieren durch Auswahl, durch Einfügen, Quicksort)
- Laufzeitanalyse der Sortierverfahren
- Datenstruktur Heap und Heap-Sort

Binäre Suche

Motivation

Ein Ratespiel

Wie kann man mit möglichst wenigen Fragen
eine Zahl $x \in \mathbb{N}$ zwischen 0 und N erraten?

Erlaubte Fragen: "Ist die Zahl kleiner als _ ?"
Antwort: "Ja / Nein"

Wie viele Fragen sind dazu notwendig?

Algorithmische Idee

- Verwende ein Intervall $[l, h)$ mit $x \in [l, h)$, das in jedem Schritt halbiert wird
- Es gilt: $l \leq x$ und $x < h$
- Anfangsintervall ist $[0, N + 1)$
- Rekursive Strategie:

  Basisfall: Wenn $h - l = 1$, dann ist $x = l$
  Rekursiver Schritt:

  - Frage, ob die Zahl kleiner ist als $m = l + (h - l)/2$
  - Falls ja, suche die Zahl im Intervall $[l, m)$
  - Andernfalls, suche die Zahl im Intervall $[m, h)$

Implementierung: Ratespiel

```
public class Ratespiel {
    // Sucht die Zahl im Interval [lo,hi)
    public static int search(int lo, int hi) {
        // Basisfall
        if ((hi-lo) == 1) {
            return lo;
        }
        // Rekursiver Schritt
        int mid = lo + (hi -lo) / 2;
        StdOut.println("Ist die Zahl kleiner als " + mid + "? (j/n)");
        if (StdIn.readString().equals("j")) {
            return search(lo,mid);
        } else {
            return search(mid,hi);
        }
    }
    public static void main(String[] args) {
        // Befehlzeilenparameter: Obere Grenze!
        int N = Integer.parseInt(args[0]);
        StdOut.println("Rate eine Zahl zwischen 0 und " + N-1 +"!");
        int x = search(0, N);
        StdOut.println("Die gesuchte Zahl ist " + x);
    }
}
```

Korrektheit des Verfahrens

Vereinfachende Annahme: $N = 2^n$ für ein $n \in \mathbb{N}$

Beobachtungen für die Aufrufe von `search(h,l)`:

- Das Intervall enthält immer die gesuchte Zahl.
- Die Intervallgrößen, d.h. die Differenz $h - l$, sind Zweierpotenzen, die mit jedem rekursiven Aufruf kleiner werden, beginnend mit 2^n und endend mit $2^0 = 1$.
  [Beweis durch Induktion]

Laufzeitanalyse: Wie viele Fragen werden benötigt?

- Nach Annahme: $N = 2^n$ für ein $n \in \mathbb{N}$, d.h. $n = \log_2(N)$.
- Sei $T(N)$ die Anzahl der Fragen.
- In jedem Rekursionsschritt wird eine Frage gestellt und das Problem auf einem Intervall halber Größe gelöst:

$$T(N) = T(N/2) + 1$$

- Im Basisfall ist keine Frage mehr nötig: $T(1) = 0$
- Insgesamt:

$$\begin{aligned} T(N) &= T(2^n) \\ &= T(2^{n-1}) + 1 = T(2^{n-2}) + 2 = \cdots = T(1) + n - 1 \\ &= n \\ &= \log_2(N) \end{aligned}$$

Binäre Suche in sortiertem Array

- Idee der Binären Suche auch beim Suchen in sortierten Datensammlungen anwendbar, die direkten Zugriff auf einen Eintrag erlauben
- Alltagsbeispiele: Wörterbuch, Telefonbuch

Problembeschreibung:
Gegeben sei eine Folge $s_0, \ldots, s_{N-1}$ von sortierten *Datensätzen*.

Jeder Datensatz s_j hat einen zugehörigen Schlüssel k_j.
Wir gehen hier davon aus, dass die Schlüssel Integer sind.

Für sortierte Datensätze gilt:

$$k_0 \leq k_1 \leq \cdots \leq k_{N-1}$$

Beschreibung der Daten

```
class DataSet {
  int key;
  String data;

  DataSet (int key, String data) {
    this.key = key;
    this.data = data;
  }
}
```

- Für bessere Übersichtlichkeit auf den Folien verzichten wir hier auf Getter/Setter-Methoden.

Implementierung

```
public class BinarySearch {
  // Liefert null, wenn Datensatz mit Index k nicht in f enthalten ist;
  // andernfalls die Referenz auf den gefundenen Datensatz
  public static DataSet search (int k, DataSet[] f) {
    return search(k, f, 0, f.length);
  }

  public static DataSet search (int k, DataSet[] f, int lo, int hi) {
    // Element nicht gefunden
    if (hi <= lo) {
        return null;
    }
    int mid = lo + (hi-lo) /2;
    if (k < f[mid].key) {
        // Suche im Intervall [lo, mid)
        search (k, f, lo, mid);
    }
    if (k > f[mid].key) {
        // Suche im Intervall [mid+1, hi)
        search (k, f, mid+1, hi);
    }
    // Element gefunden
    return f[mid];
  }
}
```


Sortieren

Sortieren

Sortieren ist eine Standardaufgabe, die Teil vieler spezieller oder auch umfassenderer Aufgabenstellungen ist.

Untersuchungen zeigen, dass „mehr als ein Viertel der kommerziell verbrauchten Rechenzeit auf Sortiervorgänge entfällt“
(Ottmann, Widmayer: Algorithmen und Datenstrukturen, Kap. 2).

Begriffsklärung: Sortierproblem

Aufgabe des **Sortierproblems** ist es, eine Permutation π zu finden, so dass die Umordnung der Datensätze gemäß π folgende Reihenfolge auf den Schlüsseln ergibt:

$$k_{\pi(0)} \leq k_{\pi(2)} \leq \cdots \leq k_{\pi(N-1)}$$

Bemerkung

Offene Aspekte der Formulierung des Sortierproblem:

- Was heißt, eine Folge ist „gegeben“?
- Ist der Bereich der Schlüssel bekannt?
- Welche Operationen stehen zur Verfügung, um π zu bestimmen?
- Was genau heißt „Umordnung“?

Vorgehen

Wir betrachten hier vier wichtige Sortieralgorithmen:

- Sortieren durch Auswählen
- Sortieren durch Einfügen
- Quicksort
- Heapsort (nächste Vorlesung)

Die algorithmische Grundidee ist unabhängig vom verwendeten Programmierparadigma.

Sortieren durch Auswahl (*Selection Sort*)

Algorithmische Idee:

- Entferne einen minimalen Eintrag `min` aus der Menge der Datensätze.
- Sortiere die Menge der Datensätze, aus der `min` entfernt wurde.
- Füge `min` als erstes Element vorne an die sortierte Ergebnismenge an.

Implementierung auf Arrays:

1. Finde einen Index *imin* des Arrays, so dass $f[imin]$ ein minimales Element von $f[0]$ bis $f[N-1]$ enthält
2. Vertausche $f[imin]$ und $f[0]$
3. Sortiere dann den Bereich $f[1]$ bis $f[N-1]$ auf gleiche Art

Implementierung: Sortieren durch Auswahl

```
static void selectionsort (DataSet[] f) {
    for (int i = 0; i < f.length - 1; i++) {
      // Finde Index fuer minimales Element
      int imin = i;
      for (int j = i+1; j < f.length; j++) {
        if (f[j].key < f[imin].key) {
          imin = j;
        }
      }
      // Vertausche Elemente
      swap(f,i,imin);
    }
}
static void swap (DataSet[] f, int i, int j) {
    DataSet tmp = f[i];
    f[i] = f[j];
    f[j] = tmp;
}
```

Laufzeitanalyse von Sortieren durch Auswahl

Kostenmodell

Wir betrachten die Anzahl der Schlüsselvergleiche C und der Zuweisungen M von Datensätzen in Abhängigkeit von der Anzahl N der Datensätze. [1]

→ pro innerem Schleifendurchlauf ein Schlüsselvergleich

→ pro äußerem Schleifendurchlauf 3 Datensatzzuweisungen

Schlüsselvergleiche: $C(N) = \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} 1 \quad \in O(N^2)$

Datensatzzuweisungen: $M(N) = \sum_{i=0}^{N-2} 3 \quad \in O(N)$

[1] Für eine detailliertere Analyse: Siehe Folien 16 "Algorithmenanalyse"

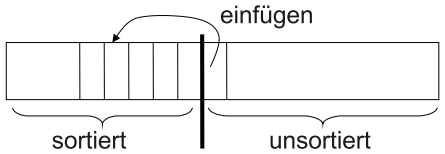
Sortieren durch Einfügen

Algorithmische Grundidee:

- Sortiere zunächst eine Teilmenge (Basisfall: leere Menge).
- Füge dann die verbleibenden Elemente nacheinander in die bereits sortierte Teilmenge an der richtigen Stelle ein.

Implementierung auf Arrays:

- Sortierter Teil des Arrays wächst in jedem Schritt um einen Eintrag
- Einfügen durch schrittweises Verschieben (ausgehend vom größten Element)



Implementierung: Sortieren durch Einfügen

```
static void insertionSort(DataSet[] f) {
  DataSet tmp; // einzufuegender Datensatz

  for (int i = 1; i < f.length; i++) {
    int j = i;
    tmp = f[j];

    // Finde neue Position fuer aktuellen Datensatz tmp
    while( j >= 1 && f[j-1].key > tmp.key ) {
       // Verschiebe Datensaetze nach hinten
       f[j] = f[j-1];
       j--;
    }
    // Setze tmp an neue Position
    f[j] = tmp;
  }
}
```


Laufzeitabschätzung I

Günstigster Fall

Array ist bereits aufsteigend sortiert

→ kein innerer Schleifendurchlauf

→ pro äußerem Schleifendurchlauf ein Schlüsselvergleich

→ pro äußerem Schleifendurchlauf zwei Datensatzzuweisungen

Schlüsselvergleiche: $C_{min}(N) = N - 1$

Datensatzzuweisungen: $M_{min}(N) = 2(N - 1)$

Laufzeitabschätzung II

Ungünstigster Fall

Array ist absteigend sortiert

→ pro äußerem Schleifendurchlauf i Schlüsselvergleiche

→ pro äußerem Schleifendurchlauf $(i+2)$ Datensatzzuweisungen

Schlüsselvergleiche: $C_{max}(N) = \sum_{i=1}^{N-1} i \quad \in O(N^2)$

Datensatzzuweisungen: $M_{max}(N) = \sum_{i=1}^{N-1} (i+2) \quad \in O(N^2)$

Quicksort

Algorithmische Grundidee

- Wähle einen beliebigen Datensatz mit Schlüssel k aus (Pivotelement)
- Teile die Datensätze in zwei Teilmengen:
  - 1. Teil enthält alle Datensätze mit Schlüsseln $< k$
  - 2. Teil enthält die Datensätze mit Schlüsseln $\geq k$
- Wende Quicksort rekursiv auf die Teillisten an
- Füge die resultierenden, nun sortierten Teillisten und das Pivotelement wieder zusammen

Beispiel: Partitionierung

17	12	6	19	23	8	5	10

Wähle als Pivotelement das letzte Element (10):

17	12	6	19	23	8	5	10

Sortiere die übrigen Elemente:

17	12	6	19	23	8	5	10

Vertausche 5 und 17:

5	12	6	19	23	8	17	10

Vertausche 8 und 12:

5	8	6	19	23	12	17	10

Keine Vertauschung mehr möglich:

5	8	6	19	23	12	17	10

Tausche nun das Pivotelement zwischen die beiden Partitionen:

5	8	6	10	23	12	17	19

Quicksort im Detail I

Implementierung auf Arrays

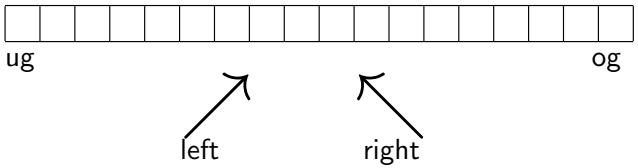
- Bearbeite rekursiv Teilbereiche des Arrays von Index `ug` bis Index `og`
- Realisiere das Teilen der Datensatzmenge durch Vertauschen
  - → Indexzähler `left`, `right` laufen von links bzw. rechts und suchen Einträge, die vertauscht werden können
  - → Für die zu tauschenden Einträge gilt: `f[left].key` $\geq$ `pivot.key` und `f[right].key` $<$ `pivot.key`
  - → In jedem Schritt gilt:

    Für alle i in [`ug`, `left`−1] : `f[i].key` $<$ `pivot.key`
    Für alle i in [`right`+1, `og`] : `pivot.key` $\leq$ `f[i].key`

Quicksort im Detail II

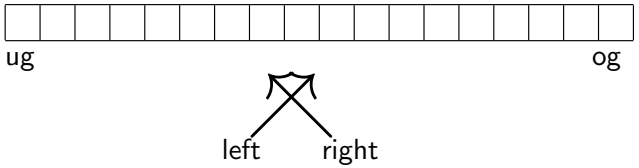
Solange `left` < `right`:

Vertausche `f[left]` und `f[right]`, falls `f[left].key` $\geq$ `pivot.key` und `f[right].key` < `pivot.key`, inkrementiere `left` und dekrementiere `right`, und fahre dann mit der Suche nach vertauschbaren Elementpaaren fort.



Quicksort im Detail III

Umsortierung des (Teil-)Arrays ist abgeschlossen, sobald `left` > `right`



Implementierung: Quicksort I

```
static void sortieren (DataSet[] f) {
  quicksort(f, 0, f.length - 1);
}

static void quicksort (DataSet[] f, int ug, int og) {
   if (ug < og) {
      int ixsplit  = partition(f, ug, og);

      /* Es gilt:
           ug ≤ ixsplit ≤ og
        && f[ixsplit].key == pivotKey
        && ( ∀ i: ug ≤ i < ixsplit
                  ⇒ f[i].key ≤ pivotKey )
        && ( ∀ i: ixsplit < i ≤ og
                  ⇒ f[i].key ≥ pivotKey )  */

      quicksort( f, ug, ixsplit-1 );
      quicksort( f, ixsplit+1, og );
   }
}
```


Implementierung: Quicksort II

```
static int partition (DataSet[] f, int ug, int og){
  int left = ug;
  int right = og - 1;

  int pivot = f[og].key;
  while (true) {
     while (f[left].key < pivot) { left++; }
     while (left <= right && f[right].key >= pivot){ right--; }
     if( left > right ) {
        break;
     } else {
        swap(f, left, right);
        left++; right--;
     }
  }
  swap(f, left, og); // Pivotelement kommt zwischen
  return left;       // die ermittelten Bereiche
}
```

Laufzeitabschätzung von Quicksort I

- Bei der Partitionierung wird während des Scans jedes Arrayelement einmal mit dem Pivot verglichen (und ggf. vertauscht).
- Anzahl der rekursiven Aufrufe und die Struktur des Aufrufbaums hängt von den Pivotelementen ab

Laufzeitabschätzung von Quicksort II

Ungünstigster Fall:
Das Pivotelement ist immer das kleinste bzw. größte Element des aktuell zu partitionierenden Teilarrays.

Beim Partitionieren ist dann jeweils eines der Teilarrays leer. Daher hat der Aufrufbaum der rekursiven Aufrufe die Tiefe N, also gilt für die Schlüsselvergleiche:

$$C_{max}(N) \in O(N^2)$$

Im ungünstigsten Fall müssen ausserdem alle Elemente bei der Partitionierung vertauscht werden:

$$M_{max}(N) \in O(N^2)$$

Laufzeitabschätzung von Quicksort III

Günstigster Fall
Das Pivotlement ist der Median des aktuell zu partitionierenden Teilarrays.

Beim der Partitionierung entstehen jeweils zwei etwa gleich große Teilarrays. Dann hat der Aufrufbaum die Tiefe $\log N$, also gilt:

$$C_{min}(N) \in O(N \log N)$$

Im günstigsten Fall muss ausserdem jedesmal das Pivotelement getauscht werden, sonst aber keine Elemente:

$$M_{min}(N) \in O(N)$$

Bemerkungen

- In der Praxis einer der schnellsten und wichtigsten Sortieralgorithmen!
- Genauere Analysen zeigen, dass Quicksort im Mittel $O(N \log N)$ Vergleiche und Tauschoperationen benötigt.
- Der ungünstigste Fall ist sehr unwahrscheinlich.

Optimierungen

- Die vorgestellte Quicksort-Fassung arbeitet schlecht auf schon sortierten Arrays.
  - → Shuffle des Arrays, um Vorsortierung aufzuheben
  - → Randomisierte Auswahl des Pivot-Elements (Beispiel: Median von 3 Elementen)
- Reduzierung der rekursiven Aufrufe durch Verwendung von InsertionSort, wenn Teil-Arrays nur noch wenige Elemente enthalten

Divide-and-Conquer Strategie

Quicksort ist ein typischer Algorithmus, der eine Divide-and-Conquer-Strategie verfolgt:

- Zerlege das Problem in Teilprobleme.
- Wende den Algorithmus rekursiv auf die Teilprobleme an.
- Füge die Teilergebnisse wieder zusammen.

Zusammenfassung

- Binäre Suche ($\Rightarrow O(\log N)$)
- Sortierverfahren auf Arrays
  - Selection Sort ($\Rightarrow O(N^2)$)
  - Insertion Sort ($\Rightarrow O(N^2)$)
  - Quicksort ($\Rightarrow O(N^2)$)

Ausblick:

- Heap Sort ($\Rightarrow O(N \log N)$)
- Abstrakte Datentypen: Stack und Queue
- Parametrisierte Datentypen in Java: Generics