Programmiertechnik II

Analyse von Algorithmen

Algorithmenentwurf

- Algorithmen sind oft Teil einer größeren Anwendung
  - operieren auf Daten der Anwendung, sollen aber unabhängig von konkreten Typen sein
    - Darstellung der Algorithmen mit Hilfe generischer Typen
  - Eigenschaften der Gesamtanwendung hängen von Auswahl des Algorithmus ab
  - Algorithmen oft nicht direkt einsetzbar
    - Aufbereitung der Daten in Eingabeformat des Algorithmus oder Anpassung des Algorithmus?
    - Behandlung spezifischer Fehlerfälle
- Ziele des Algorithmenentwurfs
  - Korrektheit
  - Universalität: Anpassbarkeit an unterschiedliche Einsatzszenarien
  - Effizienz: Suche nach Algorithmen mit geringer Laufzeit
  - Einfachheit: Lesbarkeit, Validierbarkeit, Anpassbarkeit

Empirische Analyse

- Laufzeit von verschiedenen Algorithmen wird durch Messung bestimmt
  - Vergleich nur möglich bei gleichen Testbedingungen
    - gleiche Hardware, gleiche Software (mit Ausnahme des Algorithmus), gleiche Eingabedaten
    - Ausschluss von anderen Einwirkungen
      - Maschine soll unbelastet (*idle*) sein
- Problem: Abhängigkeit des Messergebnisses von Eingabe
  - Analyse mit “echten” Daten, Zufallsdaten, Sonderfälle
- Problem: Statistische Streuung
  - Mehrfache Wiederholung des Testlaufs
  - Umfangreiche Eingabedaten
- Problem: Einfluss des Messverfahrens auf das Ergebnis
  - Analyse des Messverfahrens selbst

Häufige Fehler

1. Ignoranz von Effizienzaspekten

   Auswahl eines einfachen Algorithmus aus Angst vor Kompliziertheit des effizienten

2. Überbewertung von Effizienzaspekten

   Komplizierter (effizienter) Algorithmus wird “aus Prinzip” gewählt, selbst wenn der Algorithmus nur auf kleinen Datenmengen operieren und selten aufgerufen wird

Mathematische Analyse

- Ziele
  - Vergleich mehrerer Algorithmen für die gleiche Aufgabenstellung
  - Vorhersage der Leistung (performance) eines Algorithmus auf einem neuen Zielsystem
  - Festlegung von Parametern für einen Algorithmus
- Ideal: Definition eines exakten math. Modells der Leistung
  - math. Formeln, die Leistung in Abhängigkeit von Eingabeparametern ausdrücken
  - Komplexität: Zahl, die die Leistung ausdrückt
- Probleme:
  - Analyse ist führt zu mathematisch ungelösten Problemen
  - Analyse benötigt Informationen, die nicht bekannt sind
    - Eigenschaften von Zielcodeanweisungen (etwa JVM Bytecodes)
    - Eigenschaften der Eingabedaten

Grundlagen der Analyse

- Metriken: Messgrößen der Leistung eines Programms
  - Benötigte Rechenzeit (in Abhängigkeit von Eingabe)
  - Benötigter Hauptspeicher
  - Zahl der Speicherzugriffe
  - Zahl der Gleitkommaoperationen
  - …
- Abstrakte Messgrößen: abstrakte Operationen
  - Messgröße hängt nicht von Zielmaschine ab
    - es werden nur die "teuren" Operationen gezählt
  - Zahl der Zugriffe auf ein Feld
  - Zahl der Vergleichsoperationen
  - Ideal: tatsächliche Rechenzeit ergibt sich aus Skalierung der abstrakten Größen
  - Real: tatsächliche Rechenzeit hängt zusätzlich von anderen Faktoren ab
    - Beispiel: Speicherzugriff hängt davon ab, ob Wert im Cache steht oder im Hauptspeicher

Grundlagen der Analyse

- Zahl der Operationen hängt von Eingabe ab
  - Verallgemeinerung: Aussagen über Mengen von Eingabedaten
- Durchschnitt (average-case performance)
  - Annahme: alle Eingabedaten sind statistisch gleichverteilt
  - Annahme ist u.U. unrealistisch
- Maximalwert (worst-case performance)
  - Ermittlung des Eingabedatensatzes mit maximaler Komplexität
  - dieser Fall tritt u.U. nie ein
- Unterschied zwischen Durchschnitt und Maximum gibt Einblick in die Datenabhängigkeit der Komplexität
  - bei großen Abweichungen muss untersucht werden, unter welchen Umständen die “schlechten” Fälle auftreten

Logarithmus und ganze Zahlen

- “kleinste ganze Zahl größer als lg N”
  - Zahl der Bits, die zur Darstellung von N nötig sind

2. for(lgN = 0; N > 0; lgN++, N /= 2);
3. for(lgN = 0, t = 1; t < N; lgN++, t+=t);

$$\lfloor x \rfloor \quad \textit{größte ganze Zahl kleiner oder gleich } x$$

$$\lceil x \rceil \quad \textit{kleinste ganze Zahl größer oder gleich } x$$

Big-Oh (Landau-Notation)

Sei f: $\mathbb{N} \rightarrow \mathbb{R}$ eine Funktion. Die Menge O(f) enthält alle Funktionen g, die ab einem gewissen n_0 höchstens so schnell wachsen wie f, abgesehen von jeweils einem konstanten Faktor c:

$$O(f) = \{g : \mathbb{N} \rightarrow \mathbb{R} | \; \exists c > 0 \; \exists n_0 \in \mathbb{N} \; \forall n >= n_0 : g(n) <= c \cdot f(n)\}$$

- c, n_0: Parameter der konkreten Implementierung
  - In der Praxis oft entscheidend für die tatsächliche Rechenzeit
- O-Notation gibt nur die Größenordnung an
  - $O(n^2+n) = O(n^2)$
  - Komplexitätsklasse

Komplexitätsklassen

- Parameter ist implizit “n”:
  - $O(n) = O(f)$ mit $f(n)=n$
  - $O(1)$: konstant
  - $O(\log n)$: logarithmisch
  - $O(n)$: linear
  - …
- $O(f)$ gibt obere Schranke an
  - $8n \in O(n^2)$
  - $65 \in O(1)$

Bestimmung der asymptotischen Komplexität

- Komplexität hängt von der Eingabemenge ab
  - Eingabedaten oft iterativ oder rekursiv verarbeitet
- Schleifen: Zählen der Durchläufe, Abschätzung der Kosten eines Durchlaufs
  - evtl. Abschätzung der Zahl der Durchläufe (obere Schranke)
- Rekursion: induktive Berechnung der Komplexität
  - Berechnung des nicht-rekursiven Falls (Abbruch der Rekursion)
  - induktives Folgern der rekursiven Fälle

Beispiel: Lineare Suche

```
static int search(int a[], int v, int l, int r)
{
  int i;
  for(i=l; i <= r; i++)
   if (v == a[i])
       return i;
  return –1;
}
```

Beispiel: Binäre Suche

- Annahme: Feld ist aufsteigend sortiert

```
static int search(int a[], int v, int l, int r)
{
  while(r >= l) {
   int m = (l+r)/2;
   if (v == a[m]) return m;
   if (v < a[m]) r = m-1; else l = m+1;
  }
  return –1;
}
```