Programmiertechnik II

Ralf Herbrich

Analyse von Algorithmen

Überblick

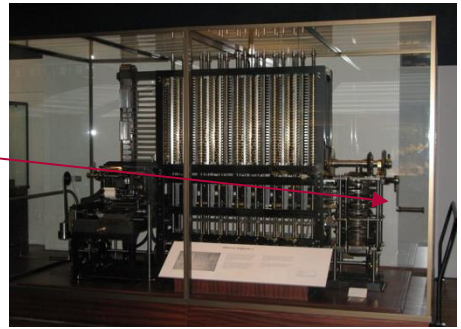
1. Wissenschaftliche Methode (*scientific method*)
2. Messung von Laufzeit
3. Mathematische Modelle für Laufzeiten
4. Klassifikation von Komplexität: Θ-, $\mathcal{O}$-, und Ω-Notation

Überblick

1. **Wissenschaftliche Methode (*scientific method*)**
2. Messung von Laufzeit
3. Mathematische Modelle für Laufzeiten
4. Klassifikation von Komplexität: Θ-, $\mathcal{O}$-, und Ω-Notation

Laufzeit

- Laufzeit von Programmen war schon immer von zentraler Bedeutung
  - *Analytic Engine*: Wie oft muss man die Kurbel für eine Berechnung drehen?



Charles Babbage (1791 – 1871)

- Laufzeit von Programmen ist für alle wichtig
  - **Programmierer**: Muss eine Softwarelösung entwickeln, die in endlicher Zeit berechnet wird
  - **Kunde**: Will die kostengünstigste Softwarelösung (Fixkosten = Softwareentwicklung, variable Kosten = Programmlaufzeitkosten)
  - **Theoretiker**: Will Algorithmen vergleichen und in Komplexitätsklassen einteilen
- Gründe, um (die Laufzeit von) Algorithmen zu analysieren
  - (Laufzeit-) Vorhersage
  - Algorithmenvergleich
  - (Laufzeit-) Garantien
  - Theoretische Basis

Programmiertechnik II (PT2)

Theoretische Informatik I (TI1)

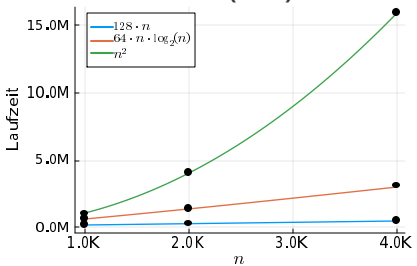
Beispiele für Algorithmische Erfolge

- ***Discrete Fourier Transform***
  - **Problem**: Zerlegung einer Zeitreihe von n Beobachtungen in periodische Komponenten
  - **Anwendung**: JPEG-Kompression, Magnet-Resonanz (MRI) Bildgebung …
  - **Naive**: Eine Doppelschleife mit n^2 Einzelschritten
  - **FFT-Algorithmus**: $n \cdot \log(n)$ Einzelschritte. Ermöglichte ganz neue Anwendungen!

- ***N*-Körper Problem**
  - **Problem**: Simulation von Gravitationskräften zwischen N Körpern
  - **Anwendung**: Astronomie
  - **Naive**: Eine Doppelschleife mit n^2 Einzelberechnungen
  - **Barnes-Hut Algorithmus**: $n \cdot \log(n)$ Einzelschritte. Ermöglichte neue Forschung!

Carl Friedrich Gauss
(1777 - 1855)

Andrew Appel
(1960)

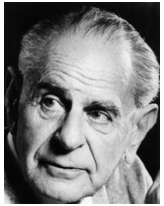


Wissenschaftliche Methode

- **Wissenschaftliche Methode**. Die wissenschaftliche Methode zeichnet sich durch 5 Schritte aus:
  1. **Beobachtung** von Merkmalen in der Welt
  2. Aufstellen einer **Hypothese**, die konsistenz mit den Beobachtungen sind
  3. **Vorhersage** von zukünftigen Beobachtungen mit Hilfe der Hypothese
  4. **Abgleichen** der Vorhersage mit der realen Beobachtung zu den Vorhersagen
  5. **Iterieren** bis Abgleich und Vorhersage übereinstimmen
- **Prinzipien**:
  1. Experimente müssen reproduzierbar sein!
  2. Hypothesen müssen falsifizierbar sein!

- Die wissenschaftliche Methode eignet sich für die Analyse von Algorithmen weil Beobachtungen leicht durch Berechnungen zu erzielen sind!

Sir Isaac Newton
(1643 – 1726)



Sir Karl Popper
(1902 – 1994)

Beispiel: 3-Summen Problem

- **3-Summen Problem**. Gegeben n unterschiedliche Ganzzahlen, bestimme wie viele Triples von Zahlen sich genau zu Null addieren!

i	a[i]
0	30
1	-40
2	-20
3	-10
4	40
5	0
6	10
7	5

a[i]	a[j]	a[k]	sum
30	-40	10	0
30	-20	-10	0
-40	40	0	0
-10	0	10	0

- Dieses Problem definiert eine ganze Komplexitätsklasse in *computational geometry*!

Anka Gajentaan (1968)

3-Summen Problem: Naive Lösung

```cpp
#include <iostream>

#define MAX_SIZE 1000000

using namespace std;

// counts the number of triples that sum to exactly 0
int count_3sums(const int* list, const int size) {
    int count = 0;
    for (auto i = 0; i < size; i++) {
        for (auto j = i + 1; j < size; j++) {
            for (auto k = j + 1; k < size; k++) {
                if (list[i] + list[j] + list[k] == 0) {
                    count++;
                }
            }
        }
    }
    return count;
}

// main entry point of the program
int main(int argc, char* argv[]) {
    int a[MAX_SIZE];
    int n = 0;

    // read the list of integers from the standard input
    while (n < MAX_SIZE && (cin >> a[n])) {
        n++;
    }

    // computes the number of triples that sum to exactly 0 and outputs the count on the screen
    cout << count_3sums(a, n) << endl;

    return (0);
}
```

Iteriere alle `a[i]`

Iteriere alle `a[j]`

Iteriere alle `a[k]`

Überprüfe, dass `a[i]+ a[j]+ a[k] == 0`

Überblick

1. Wissenschaftliche Methode (*scientific method*)
2. **Messung von Laufzeit**
3. Mathematische Modelle für Laufzeiten
4. Klassifikation von Komplexität: Θ-, $\mathcal{O}$-, und Ω-Notation

Messmethoden

- Manuell mit Stoppuhr

```
● ➜  unit3 git:(main) ✗ ./3sums < 1Kints.txt
  70
● ➜  unit3 git:(main) ✗ ./3sums < 2Kints.txt
  528
● ➜  unit3 git:(main) ✗ ./3sums < 4Kints.txt
  4039
● ➜  unit3 git:(main) ✗ ./3sums < 8Kints.txt
  32074
○ ➜  unit3 git:(main) ✗
```

- Mit Hilfe des UNIX-Werkzeugs `time`

```
● ➜  unit3 git:(main) ✗ time ./3sums < 1Kints.txt
  70
  ./3sums < 1Kints.txt  0.02s user 0.00s system 94% cpu 0.028 total
● ➜  unit3 git:(main) ✗ time ./3sums < 2Kints.txt
  528
  ./3sums < 2Kints.txt  0.11s user 0.00s system 97% cpu 0.116 total
● ➜  unit3 git:(main) ✗ time ./3sums < 4Kints.txt
  4039
  ./3sums < 4Kints.txt  0.67s user 0.00s system 98% cpu 0.680 total
● ➜  unit3 git:(main) ✗ time ./3sums < 8Kints.txt
  32074
  ./3sums < 8Kints.txt  4.92s user 0.00s system 98% cpu 4.979 total
○ ➜  unit3 git:(main) ✗
```

Messmethoden (*ctd.*)

- Mit Hilfe von Standardbibliotheken in `ctime.h`

```
#include <iostream>
#include <iomanip>
#include <ctime>

#define MAX_SIZE 1000000

using namespace std;

// counts the number of triples that sum to exactly 0
int count_3sums(const int* list, const int size) {
...
}

// main entry point of the program
int main(int argc, char* argv[]) {
    int a[MAX_SIZE];
    int n = 0;

    // read the list of integers from the standard input
    while (n < MAX_SIZE && (cin >> a[n])) {
        n++;
    }

    // get the initial clock count
    auto start = clock();

    // computes the number of triples that sum to exactly 0
    cout << count_3sums(a, n) << endl;

    // output execution time
    cout << "Time of execution: "
         << setprecision(4)
         << double(clock() - start)/double(CLOCKS_PER_SEC)
         << " seconds" << endl;

    return (0);
}
```

Ermittle momentanen Taktzähler

Lasse den Algorithmus laufen

Ermittle Taktdifferenz und normiere auf Sekunden

Empirische Analyse

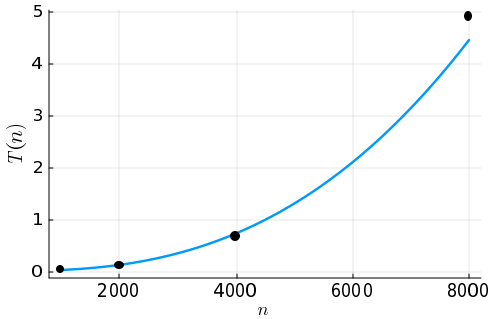
4 Läufe des Programms

n	Laufzeit (in s)
1000	0.02104
2000	0.107
4000	0.6583
8000	4.904

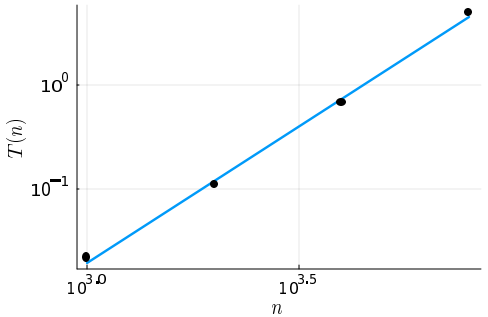
$b = 2.62$

$2^c = 2^{-31.8}$

Standard Plot



Log-Log Plot



power law

$T(n) = n^b \cdot 2^c$

$\log_2(T(n)) = b \cdot \log_2(n) + c$

Vorhersage

- **Hypothese**: Die Laufzeit von 3 Summen ist ungefähr $2^{-31.8} \cdot n^{2.62}$

Die Größenordnung ist ungefähr n^3

- **Vorhersage**:
  - Für $n = 8000$ ist $\hat{T}(n) = 4.457$
  - Für $n = 16000$ ist $\hat{T}(n) = 27.43$

- **Beobachtungen**

n	Laufzeit (in s)
8000	4.887
8000	4.891
16000	37.93
16000	38.02

Unterstützt die Hypothese nicht

Doubling Hypothesis

- Einfache Methode, um den Exponenten in einem *power law* zu schätzen.
- **Idee**: Lasse das Programm laufen und verdoppele die Eingabegröße

n	Laufzeit (in s)	$\frac{T(2n)}{T(n)}$	$\hat{b} = \log_2\left(\frac{T(2n)}{T(n)}\right)$
1000	0.02104		
2000	0.107	5.08	2.344
4000	0.6583	6.15	2.621
8000	4.904	7.44	2.900

$$\frac{T(2n)}{T(n)} = \frac{2^c \cdot (2n)^b}{2^c \cdot n^b} = 2^b$$

- **Schätzung von** c: Für ein geschätztes $\hat{b}$, kann c geschätzt werden mittels

$$\hat{c} = \log_2\left(\frac{T(n)}{n^{\hat{b}}}\right)$$

Konvergiert gegen 3.0

$$\hat{c} = \log_2\left(\frac{4.904}{8000^{2.9}}\right) = -35.31 \qquad \hat{T}(16000) = 2^{-35.31} \cdot 16000^{2.9} = 36.52$$

Experimentelle Algorithmik

- **Systemabhängige** Effekte, die Laufzeit bestimmen:
  - Hardware: CPU, Speicher, Cache, …
  - Software: Compiler, Interpreter, Speichermanager, …
  - System: Betriebssystem, Netzwerk, (andere) Anwendungen, …

Bestimmen die Konstante 2^c im *power law*

- **Systemunabhängige** Effekte, die Laufzeit bestimmen:
  - Eingabedaten
  - Algorithmus

Bestimmen den Exponenten b im *power law*

- Problematisch, exakte Messungen der Laufzeit zu bekommen.
- Aber, viel einfacher und kostengünstiger mehr Messungen als in anderen Wissenschaften zu machen!

Zum Beispiel, sehr viele Experimente/Ausführungen machen!

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Viel Spaß bis zur nächsten Vorlesung!