Datenstrukturen und Algorithmen

Vorlesung am D-Math (CSE) der ETH Zürich

Felix Friedrich

FS 2018

1. Einführung

Algorithmen und Datenstrukturen, drei Beispiele

Ziele der Vorlesung

- Verständnis des Entwurfs und der Analyse grundlegender Algorithmen und Datenstrukturen.
- Vertiefter Einblick in ein modernes Programmiermodell (mit C++).
- Wissen um Chancen, Probleme und Grenzen des parallelen und nebenläufigen Programmierens.

Ziele der Vorlesung

Einerseits

- Unverzichtbares Grundlagenwissen aus der Informatik.

Andererseits

- Vorbereitung für Ihr weiteres Studium und die Praxis.

Inhalte der Vorlesung

Datenstrukturen / Algorithmen

Begriff der Invariante, Kostenmodell, Landau Symbole
Algorithmenentwurf, Induktion
Suchen und Auswahl, Sortieren
Dynamic Programming
Wörterbücher: Hashing und Suchbäume, AVL
Graphen, Kürzeste Wege, Backtracking, Flow
Geometrische Algorithmen, High Performance LA
Randomisierte Algorithmen (Gibbs/SA), Multiskalen
Sortiernetzwerke, parallele Algorithmen

Programmieren mit C++

RAII, Move Konstruktion, Smart Pointers, Constexpr, user defined literals
Templates und Generische Programmierung
Exceptions
Funktoren und Lambdas
Threads, Mutexs and Monitors
Promises and Futures

Parallel Programming

Parallelität vs. Concurrency, Speedup (Amdahl/-Gustavson), Races, Memory Reordering, Atomic Registers, RMW (CAS,TAS), Deadlock/Starvation

1.2 Algorithmen

[Cormen et al, Kap. 1;Ottman/Widmayer, Kap. 1.1]

Algorithmus

Algorithmus: wohldefinierte Berechnungsvorschrift, welche aus Eingabedaten (*input*) Ausgabedaten (*output*) berechnet.

Beispielproblem

Input : Eine Folge von n Zahlen $(a_1, a_2, \ldots, a_n)$

Output : Eine Permutation $(a'_1, a'_2, \ldots, a'_n)$ der Folge $(a_i)_{1\leq i\leq n}$, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Mögliche Eingaben

$(1, 7, 3)$, $(15, 13, 12, -0.5)$, (1) ...

Jedes Beispiel erzeugt eine *Probleminstanz*.

Beispiele für Probleme in der Algorithmik

- Tabellen und Statistiken: Suchen, Auswählen und Sortieren
- Routenplanung: Kürzeste Wege Algorithmus, Heap Datenstruktur
- DNA Matching: Dynamic Programming
- Fabrikationspipeline: Topologische Sortierung
- Autovervollständigung: Wörterbücher/Bäume
- Symboltabellen: Hash-Tabellen
- Der Handlungsreisende: Dynamische Programmierung, Minimal aufspannender Baum, Simulated Annealing,
- Zeichnen am Computer: Linien und Kreise Digitalisieren, Füllen von Polygonen
- PageRank: (Markov-Chain) Monte Carlo ...

Charakteristik

- Extrem grosse Anzahl potentieller Lösungen
- Praktische Anwendung

Datenstrukturen

- Organisation der Daten, zugeschnitten auf die Algorithmen die auf den Daten operieren
- Programme = Algorithmen + Datenstrukturen.

Sehr schwierige Probleme

- NP-vollständige Probleme: Keine bekannte effiziente Lösung (Fehlen einer solchen Lösung ist aber unbewiesen!)
- Beispiel: Travelling Salesman Problem

Ein Traum

- Wären Rechner unendlich schnell und hätten unendlich viel Speicher ...
- ... dann bräuchten wir die Theorie der Algorithmen (nur) für Aussagen über Korrektheit (incl. Terminierung).

Die Realität

Ressourcen sind beschränkt und nicht umsonst:

- Rechenzeit $\rightarrow$ Effizienz
- Speicherplatz $\rightarrow$ Effizienz

1.3 Altägyptische Multiplikation

Altägyptische Multiplikation

Altägyptische Multiplikation[1]

Berechnung von $11 \cdot 9$

11	9
~~22~~	~~4~~
~~44~~	~~2~~
88	1
99	–

9	11
18	5
~~36~~	~~2~~
72	1
99	

1. Links verdoppeln, rechts ganzzahlig halbieren.
2. Gerade Zahl rechts $\Rightarrow$ Zeile streichen.
3. Übrige Zeilen links addieren.

[1] Auch bekannt als Russiche Bauernmulltiplikation

Vorteile

- Kurze Beschreibung, einfach zu verstehen.
- Effizient für Computer im Dualsystem: Verdoppeln = Left Shift, Halbieren = Right Shift

Beispiel

left shift $9 = 01001_2 \rightarrow 10010_2 = 18$
right shift $9 = 01001_2 \rightarrow 00100_2 = 4$

Fragen

- Funktioniert das immer? (z.B. für negative Zahlen)
- Wenn nicht, wann?
- Wie beweist man die Korrektheit?
- Besser als die "Schulmethode"?
- Was heisst "gut"? Lässt sich Güte anordnen?
- Wie schreibt man das Verfahren unmissverständlich auf?

Beobachtung

Wenn $b > 1$, $a \in \mathbb{Z}$, dann:

$$a \cdot b = \begin{cases} 2a \cdot \frac{b}{2} & \text{falls } b \text{ gerade,} \\ a + 2a \cdot \frac{b-1}{2} & \text{falls } b \text{ ungerade.} \end{cases}$$

Terminierung

$$a \cdot b = \begin{cases} a & \text{falls } b = 1, \\ 2a \cdot \frac{b}{2} & \text{falls } b \text{ gerade,} \\ a + 2a \cdot \frac{b-1}{2} & \text{falls } b \text{ ungerade.} \end{cases}$$

Rekursiv funktional notiert

$$f(a,b) = \begin{cases} a & \text{falls } b = 1, \\ f(2a, \frac{b}{2}) & \text{falls } b \text{ gerade,} \\ a + f(2a, \frac{b-1}{2}) & \text{falls } b \text{ ungerade.} \end{cases}$$

Funktion programmiert

```
// pre: b>0
// post: return a*b
int f(int a, int b){
    if(b==1)
        return a;
    else if (b%2 == 0)
        return f(2*a, b/2);
    else
        return a + f(2*a, (b-1)/2);
}
```


Korrektheit

$$f(a,b) = \begin{cases} a & \text{falls } b = 1, \\ f(2a, \frac{b}{2}) & \text{falls } b \text{ gerade,} \\ a + f(2a \cdot \frac{b-1}{2}) & \text{falls } b \text{ ungerade.} \end{cases}$$

Zu zeigen: $f(a,b) = a \cdot b$ für $a \in \mathbb{Z}$, $b \in \mathbb{N}^+$.

Beweis per Induktion

Anfang: $b = 1 \Rightarrow f(a,b) = a = a \cdot 1$.
Hypothese: $f(a,b') = a \cdot b'$ für $0 < b' \leq b$
Schritt: $f(a, b+1) \stackrel{!}{=} a \cdot (b+1)$

$$f(a,b+1) = \begin{cases} f(2a, \overbrace{\frac{b+1}{2}}^{\leq b}) = a \cdot (b+1) & \text{falls } b \text{ ungerade,} \\ a + f(2a, \underbrace{\frac{b}{2}}_{\leq b}) = a + a \cdot b & \text{falls } b \text{ gerade.} \end{cases}$$

■

Endrekursion

Die Rekursion lässt sich *endrekursiv* schreiben

```
// pre: b>0
// post: return a*b
int f(int a, int b){
  if(b==1)
    return a;
  else if (b%2 == 0)
    return f(2*a, b/2);
  else
    return a + f(2*a, (b-1)/2);
}
```

```
// pre: b>0
// post: return a*b
int f(int a, int b){
  if(b==1)
    return a;
  int z=0;
  if (b%2 != 0){
    --b;
    z=a;
  }
  return z + f(2*a, b/2);
}
```


Endrekursion ⇒ Iteration

```
// pre: b>0
// post: return a*b
int f(int a, int b){
  if(b==1)
    return a;
  int z=0;
  if (b%2 != 0){
    --b;
    z=a;
  }
  return z + f(2*a, b/2);
}
```

```
int f(int a, int b) {
  int res = 0;
  while (b != 1) {
    int z = 0;
    if (b % 2 != 0){
      --b;
      z = a;
    }
    res += z;
    a *= 2; // neues a
    b /= 2; // neues b
  }
  res += a; // Basisfall b=1
  return res;
}
```


Vereinfachen

```
int f(int a, int b) {
  int res = 0;
  while (b != 1) {
    int z = 0;
    if (b % 2 != 0){
      --b;  ⟶ Teil der Division
      z = a; ⟶ Direkt in res
    }
    res += z;
    a *= 2;
    b /= 2;
  }
  res += a; ⟶ in den Loop
  return res;
}
```

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0)
      res += a;
    a *= 2;
    b /= 2;
  }
  return res;
}
```


Invarianten!

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      --b;
    }
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Sei $x := a \cdot b$.

Hier gilt $x = \boxed{a \cdot b + res}$

Wenn hier $x = a \cdot b + res$ ...

... dann auch hier $x = a \cdot b + res$
b gerade

Hier gilt $x = a \cdot b + res$
Hier gilt $x = a \cdot b + res$ und $b = 0$
Also $res = x$.

Zusammenfassung

Der Ausdruck $a \cdot b + res$ ist eine *Invariante*.

- Werte von a, b, res ändern sich, aber die Invariante bleibt ”im Wesentlichen“ unverändert:
- Invariante vorübergehend durch eine Anweisung zerstört, aber dann darauf wieder hergestellt.
- Betrachtet man solche Aktionsfolgen als atomar, bleibt der Wert tatsächlich invariant
- Insbesondere erhält die Schleife die Invariante (*Schleifeninvariante*), wirkt dort wie der Induktionsschritt bei der vollständigen Induktion
- Invarianten sind offenbar mächtige Beweishilfsmittel!

Weiteres Kürzen

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      --b;
    }
    a *= 2;
    b /= 2;
  }
  return res;
}
```

$\longrightarrow$

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    res += a * (b%2);
    a *= 2;
    b /= 2;
  }
  return res;
}
```


Analyse

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    res += a * (b%2);
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Altägyptische Multiplikation entspricht der Schulmethode zur Basis 2.

$$
\begin{array}{cccccccccc}
1 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 & \\
\hline
 & & & & & 1 & 0 & 0 & 1 & (9) \\
 & & & & 1 & 0 & 0 & 1 & & (18) \\
\hline
 & & & & 1 & 1 & 0 & 1 & 1 & \\
 & & 1 & 0 & 0 & 1 & & & & (72) \\
\hline
 & & 1 & 1 & 0 & 0 & 0 & 1 & 1 & (99)
\end{array}
$$

Effizienz

Frage: Wie lange dauert eine Multiplikation von a und b?

- Mass für die Effizienz
  - Gesamtzahl der elementaren Operationen: Verdoppeln, Halbieren, Test auf "gerade", Addition
  - Im rekursiven wie im iterativen Code: maximal 6 Operationen pro Aufruf bzw. Durchlauf
- Wesentliches Kriterium:
  - Anzahl rekursiver Aufrufe oder
  - Anzahl Schleifendurchläufe(im iterativen Fall)
- $\frac{b}{2^n} \leq 1$ gilt für $n \geq \log_2 b$. Also nicht mehr als $6\lceil \log_2 b \rceil$ elementare Operationen.

1.4 Schnelle Multiplikation von Zahlen

[Ottman/Widmayer, Kap. 1.2.3]

Beispiel 2: Multiplikation grosser Zahlen

Primarschule:

$$
\begin{array}{cccccc|l}
 & a & b & & c & d & \\
 & 6 & 2 & \cdot & 3 & 7 & \\
\hline
 & & & & 1 & 4 & d \cdot b \\
 & & & 4 & 2 & & d \cdot a \\
 & & & & 6 & & c \cdot b \\
 & & 1 & 8 & & & c \cdot a \\
\hline
= & & 2 & 2 & 9 & 4 &
\end{array}
$$

$2 \cdot 2 = 4$ einstellige Multiplikationen. $\Rightarrow$ Multiplikation zweier n-stelliger Zahlen: n^2 einstellige Multiplikationen

Beobachtung

$$
\begin{aligned}
ab \cdot cd &= (10 \cdot a + b) \cdot (10 \cdot c + d) \\
&= 100 \cdot a \cdot c + 10 \cdot a \cdot c \\
&\quad + 10 \cdot b \cdot d + b \cdot d \\
&\quad + 10 \cdot (a - b) \cdot (d - c)
\end{aligned}
$$

Verbesserung?

$$
\begin{array}{cccccc|l}
 & a & b & & c & d & \\
 & 6 & 2 & \cdot & 3 & 7 & \\
\hline
 & & & & 1 & 4 & d \cdot b \\
 & & & 1 & 4 & & d \cdot b \\
 & & & 1 & 6 & & (a - b) \cdot (d - c) \\
 & & & 1 & 8 & & c \cdot a \\
 & & 1 & 8 & & & c \cdot a \\
\hline
= & & 2 & 2 & 9 & 4 &
\end{array}
$$

$\rightarrow 3$ einstellige Multiplikationen.

Grosse Zahlen

$$6237 \cdot 5898 = \underbrace{62}_{a'}\underbrace{37}_{b'} \cdot \underbrace{58}_{c'}\underbrace{98}_{d'}$$

Rekursive / induktive Anwendung: $a' \cdot c'$, $a' \cdot d'$, $b' \cdot c'$ und $c' \cdot d'$ wie oben berechnen.

$\rightarrow 3 \cdot 3 = 9$ statt 16 einstellige Multiplikationen.

Verallgemeinerung

Annahme: zwei n-stellige Zahlen, $n = 2^k$ für ein k.

$$\begin{aligned}(10^{n/2}a + b) \cdot (10^{n/2}c + d) = 10^n \cdot a \cdot c + 10^{n/2} \cdot a \cdot c \\ + 10^{n/2} \cdot b \cdot d + b \cdot d \\ + 10^{n/2} \cdot (a - b) \cdot (d - c)\end{aligned}$$

Rekursive Anwendung dieser Formel: Algorithmus von Karatsuba und Ofman (1962).

Analyse

$M(n)$: Anzahl einstelliger Multiplikationen.

Rekursive Anwendung des obigen Algorithmus $\Rightarrow$ Rekursionsgleichung:

$$M(2^k) = \begin{cases} 1 & \text{falls } k = 0, \\ 3 \cdot M(2^{k-1}) & \text{falls } k > 0. \end{cases}$$

Teleskopieren

Iteratives Einsetzen der Rekursionsformel zum Lösen der Rekursionsgleichung.

$$\begin{aligned}M(2^k) &= 3 \cdot M(2^{k-1}) = 3 \cdot 3 \cdot M(2^{k-2}) = 3^2 \cdot M(2^{k-2}) \\ &= \ldots \\ &\overset{!}{=} 3^k \cdot M(2^0) = 3^k.\end{aligned}$$

Beweis: Vollständige Induktion

Hypothese H:

$$M(2^k) = 3^k.$$

Induktionsanfang ($k = 0$):

$$M(2^0) = 3^0 = 1. \quad \checkmark$$

Induktionsschritt ($k \to k + 1$):

$$M(2^{k+1}) \overset{\text{def}}{=} 3 \cdot M(2^k) \overset{\text{H}}{=} 3 \cdot 3^k = 3^{k+1}.$$

■

Vergleich

Primarschulmethode: n^2 einstellige Multiplikationen.

Karatsuba/Ofman:

$$M(n) = 3^{\log_2 n} = (2^{\log_2 3})^{\log_2 n} = 2^{\log_2 3 \log_2 n} = n^{\log_2 3} \approx n^{1.58}.$$

Beispiel: 1000-stellige Zahl: $1000^2/1000^{1.58} \approx 18$.

Bestmöglicher Algorithmums?

Wir kennen nun eine obere Schranke $n^{\log_2 3}$.

Es gibt praktisch (für grosses n) relevante, schnellere Algorithmen. Die beste obere Schranke ist nicht bekannt.

Untere Schranke: $n/2$ (Jede Ziffer muss zumindest einmal angeschaut werden).

1.5 Finde den Star

Konstruktiv?

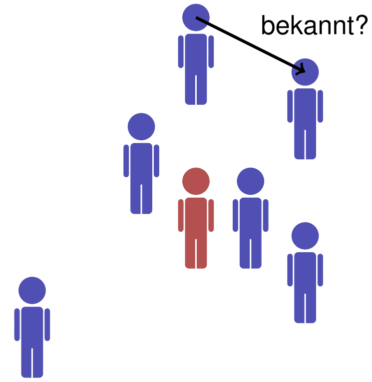
Übung: Finde ein schnelleres Multiplikationsverfahren.
Unsystematisches Suchen einer Lösung ⇒ ☹.

Betrachten nun ein konstruktiveres Beispiel.

Beispiel 3: Finde den Star!

Raum mit $n > 1$ Personen.

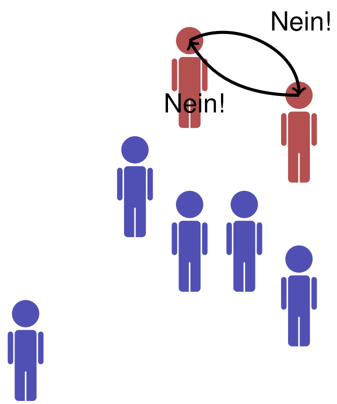
- *Star:* Person, die niemanden anderen kennt, jedoch von allen gekannt wird.
- *Elementare Operation:* Einzige erlaubte Frage an Person A: "Kennst Du B?" ($B \neq A$)



Problemeigenschaften

- Möglich: kein Star anwesend.
- Möglich: ein Star.
- Mehr als ein Star möglich?

Annahme: Zwei Stars S_1, S_2.
S_1 kennt S_2 ⇒ S_1 kein Star.
S_1 kennt S_2 nicht ⇒ S_2 kein Star. $\perp$



Naive Lösung

Frage jeden über jeden.

Resultat:

	1	2	3	4
1	-	Ja	Nein	Nein
2	Nein	-	Nein	Nein
3	Ja	Ja	-	Nein
4	Ja	Ja	Ja	-

Star ist 2.

Anzahl Operationen (Fragen): $n \cdot (n-1)$.

Geht das besser?

Induktion: zerlege Problem in kleinere Teile.

- $n = 2$: Zwei Fragen genügen.
- $n > 2$: Schicke eine Person A weg. Finde den Star unter $n-1$ Personen. Danach überprüfe A mit $2 \cdot (n-1)$ Fragen.

Gesamt
$F(n) = 2(n-1)+F(n-1) = 2(n-1)+2(n-2)+\cdots+2 = n(n-1)$.

Kein Gewinn.

Verbesserung

Idee: Vermeide, den Star rauszuschicken.

- Frage eine beliebige Person A im Raum, ob sie B kennt.
- Falls ja: A ist kein Star.
- Falls nein: B ist kein Star.
- Zum Schluss bleiben 2 Personen, von denen möglicherweise eine Person X der Star ist. Wir überprüfen mit jeder Person, die draussen ist, ob X ein Star sein kann.

Analyse

$$F(n) = \begin{cases} 2 & \text{für } n = 2, \\ 1 + F(n-1) + 2 & \text{für } n > 2. \end{cases}$$

Teleskopieren:

$$F(n) = 3+F(n-1) = 2\cdot 3+F(n-2) = \cdots = 3\cdot(n-2)+2 = 3n-4.$$

Beweis: Übung!

Moral

Bei vielen Problemen lässt sich ein induktives oder rekursives Lösungsmuster erarbeiten, welches auf der stückweisen Vereinfachung eines Problems basiert. Weiteres Beispiel in der nächsten Stunde.