Freie Universität Berlin

Proseminar Theoretische Informatik

Zeitkomplexität I

Einführung und die Komplexitätsklasse P

26.11.2010, Sebastian Stugk

Organisatorisches

1. Fragen

2. Mitschreiben, oder nicht?

3. äquivalent verwendete Begriffe

   Entscheidung ↔ Lösung ↔ Berechnung

   Turingmaschine ↔ Algorithmus

   Problem ↔ Sprache

   Laufzeit ↔ Zeitkomplexität

Was euch erwartet

1. Grundlegendes zu Zeitkomplexität
   - Was ist Zeitkomplexität?
   - Wie bestimmt man sie?

2. Klassifizierung von Problemen anhand ihrer Zeitkomplexität
   - Was ist eine Zeitkomplexitätsklasse?
   - Welche Abhängigkeiten vom Maschinenmodell gibt es?

3. Die Komplexitätsklasse P
   - Was ist P?
   - Welche Probleme sind in dieser Klasse enthalten, welche nicht?

4. Ein (kleiner) Blick über den Tellerrand

Ein einführendes Beispiel

Angenommen...

CPU-Leistung: 3 GHz

nach Moore's Gesetz:

Problem P und 2 Algorithmmen, die P unterschiedlich "effizent" lösen (bei Eingabelänge *n*):

- Algorithmus A1 benötigt

  n^2 Taktschritte

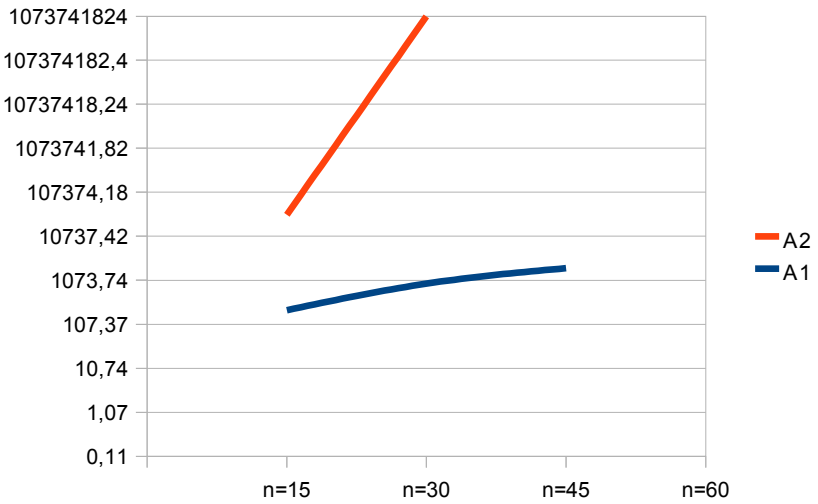
- Algorithmus A2 benötigt

  2^n Taktschritte

ca. 22 Jahre bis A2 das Problem P allein durch bessere Hardware in "nur" 1 Tag lösen kann.

Bei Eingabelänge n=60:

- A1 benötigt ca. 833 ms
- A2 benötigt ca. 12.186 Jahre



1. Grundlegendes

Was ist Zeitkomplexität?

Entscheidungsverfahren benötigen gewisse Ressourcen:

- Speicher
- Zeit

Komplexität in unserem Zusammenhang (*informal*):

- Wieviele Ressourcen sind nötig um eine Instanz eines Problems zu lösen im Verhältnis zur "Schwere" dieser Instanz?

Zeitkomplexität:

- Wir bewerten Entscheidungsverfahren nach ihrem Zeitbedarf.

Zeitkomplexität bestimmen

Wie?

- Analyse

Unter welchen Gesichtspunkten?

- keine problemspezifischen Einflussfaktoren
- einziges Kriterium:

  Länge der Eingabe

Welches Maß?

- abhängig vom zugrunde liegenden Berechnungsmodell
- hier (Turingmaschine):

  Anzahl der Kopfbewegungen

Etwas formaler

Definition:

Die **Zeitkomplexität** einer deterministischen Turingmaschine *M*, die auf allen Eingaben hält, ist eine Funktion

$$f: \mathbb{N} \rightarrow \mathbb{N}$$

wobei $f(n)$ die die Maximalanzahl von Schritten ist, die *M* bei einer Eingabe der Länge *n* macht.

gesprochen: *"M ist f(n)-Zeit-Turingmaschine"* oder *"M läuft in f(n) Zeit"*

Also:

- Wir untersuchen nur entscheidbare Probleme.
- Bei unserer Analyse betrachten wir den "worst case"-Fall.

Analyse

asymptotische Analyse:

- Untersuchung der Laufzeit eines Algorithmus bei großen Eingaben
- O-Notation

Abschätzung der Laufzeit, statt exakter Bestimmung:

- Exakte Laufzeit kann sehr komplexer mathematischer Ausdruck sein.
- Differenzen bis zu einer gewissen Größe können vernachlässigt werden.
  - Sie haben auf das asymptotische Wachstum keinen Einfluss.

Warum so "ungenau"?

- Ziel heute: Klassifizieren/Vergleichen, nicht optimieren

Asymptotische Analyse & O-Notation

Definitionen für den heutigen Vortrag

für Funktionen $f, g : \mathbb{N} \longrightarrow \mathbb{R}^+$

g ist **obere Schranke** von *f*, $\rightarrow \quad f(n) = O(g(n))$

falls konstante natürliche Zahlen *c* und n_0 existieren, sodass

$$\forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

g **starke obere Schranke** von *f*, $\rightarrow \quad f(n) = o(g(n))$

falls für alle reellen *c* und eine natürliche Zahl n_0 existiert, sodass:

$$\forall n \geq n_0 : f(n) < c \cdot g(n)$$

Asymptotische Analyse & O-Notation

anschaulicher:

1. Nur Terme höchster Ordnung werden beachtet:

   Für $f(n) = 6n^3 + 11n^2 + 3n + 77$ ist das $6n^3$

2. Konstante Faktoren werden nicht außer Acht gelassen:

   Im Fall von $6n^3$ ist n^3 *dominante* Größe.

Asymptotische Analyse & O-Notation

Die O-Notation ist verträglich mit gewöhnlicher Arithmetik:

1. Multiplikation mit einer Konstanten *k*:

$$k \cdot O(f(n)) = O(k \cdot f(n))$$

2. Addition:

Falls: $f_1(n) = O(g_1(n)) \text{ und } f_2(n) = O(g_2(n))$

Dann: $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$

Asymptotische Analyse & O-Notation

Für die folgenden Beispiele außerdem wichtig:

1. Logarithmen verschiedener Basen haben asymptotisch gleiches Wachstum.

2. Häufig auftretende Funktionen aufsteigend nach ihrem asymptotischen Wachstum sortiert:

$$O(1) = O(log(n)) = O(n) = O(n \cdot log(n)) =$$
$$O(n^2) = O(n^3) = ... = O(n^k) = O(2^n) = O(n!) = O(n^n)$$

Konventionen für die Analyse

durch O-Notation möglich:

- Algorithmusbeschreibung ohne modellspezifische Details (z.B. Kopfbewegungen der Turingmaschine)
- Formulierung der Beschreibung in menschlicher Sprache

Aber: Unsere Auslassungen dürfen die Gesamtlaufzeit um nicht mehr als eine konstanten Faktor beeinflussen

Konventionen für die Analyse

zeilenweise Gliederung:

- Eine Zeile sollte einer logisch zusammenhängenden "Phase" des Algorithmus entsprechen.

- **Wichtig:** Um eine bestimmte Laufzeitbeschränkung zu zeigen...

  Jede "Zeile" muss sich mit einer solchen Laufzeit auf dem entsprechenden Maschinenmodell implementieren lassen.

→ Dann: Aus ermittelten Laufzeiten der einzelnen Zeilen auf Gesamtlaufzeit schließen

Ein Beispiel

Im folgenden betrachten wir beispielhaft folgende Sprache:

$$A = \{0^k 1^k \mid k \geq 0\}$$

A ist kontextfrei. $\rightarrow$ *A ist entscheidbar.*

Turingmaschine M_1

Der erste Vorschlag für eine Turingmaschine, die A entscheidet:

Turingmaschine **M_1**:

"Auf Eingabe w:

1. Laufe über das Band. Verwerfe, falls eine 0 rechts von einer 1 gefunden wird.
2. Solange sowohl Nullen als auch Einsen auf dem Band verbleiben:
3. Laufe über das Band und lösche jeweils genau eine 1 genau eine 0.
4. Falls entweder Einsen oder Nullen auf dem Band verblieben sind: Verwerfe
   Sonst (falls alle Zeichen vom Band gelöscht wurden): akzeptiere."

Analyse ...

Ergebnis der Analyse von M_1

Laufzeiten der Zeilen:

1. **O(n)**
+ 2. **O(n/2)**
  * 3. **O(n)**
+ 4. **O(n)**

Gesamtlaufzeit: $O(n) + O(n/2)*O(n) + O(n)$

$= O(n) + O(n^2/2) + O(n)$

$= \mathbf{O(n^2)}$

Turingmaschine M_2

Ein Vorschlag für eine bessere Laufzeit:

Turingmaschine **M_2**:

"Auf Eingabe w

1. Laufe über das Band und verwerfe, falls ein 0 rechts von einer 1 gefunden wird.
2. Solange sowohl Nullen als auch Einsen auf dem Band verbleiben:
3. Prüfe, die Gesamtanzahl von Nullen und Einsen.
   Falls ungerade: verwerfe.
4. Beginnend mit der jeweils ersten: Markiere erst jede zweite 0 und dann jede zweite 1.
5. Falls keine Nullen und Einsen auf dem Band verbleiben: Akzeptiere.
   Sonst: Verwerfe."

zur Korrektheit an der Tafel...

Arbeitet M_2 korrekt?

Zu Zeigen war: M_2 entscheidet A.

Eckdaten des Beweises:

Fall $w \in A$:

- korrekte Reihenfolge wird durch Zeile 1 garantiert
- Streichen von 0en und 1en immer im selben Verhältnis

Fall $w \notin A$:

- nicht korrekte Reihenfolge: Verwerfen in Zeile 1
- ungerade Differenz von 0en und 1en: Verwerfen in Zeile 3
- für $w = 0^k\alpha1^k$ bzw. $w = 0^k1^k\alpha$ mit $\alpha = \{0\}^*$ bzw. $\alpha = \{1\}^*$:
  - $|\alpha| \leq \alpha$: nach weniger als $\lfloor \log_2 k \rfloor + 1$ Iterationen wird $|\alpha| = 1$, spätestens dann Verwerfen in Zeile 3
  - $|\alpha| > k$: α verbleibt als Rest nach $\lfloor \log_2 k \rfloor + 1$ Schleifendurchläufen, spätestens dann Verwerfen in Zeile 5

Analyse von M_2 ...

nochmal zum Nachvollziehen

Turingmaschine $\mathbf{M_2}$:

"Auf Eingabe w

1. Laufe über das Band und verwerfe, falls ein 0 rechts von einer 1 gefunden wird.
2. Solange sowohl Nullen als auch Einsen auf dem Band verbleiben:
3. Prüfe, die Gesamtanzahl von Nullen und Einsen.
   Falls ungerade: verwerfe.
4. Beginnend mit der jeweils ersten: Markiere erst jede zweite 0 und dann jede zweite 1.
5. Falls keine Nullen und Einsen auf dem Band verbleiben: Akzeptiere.
   Sonst: Verwerfe."

Ergebnis der Analyse von M_2

Laufzeiten der Zeilen:

1. **O(n)**
+ 2. **höchstens ($\log_2 n + 1$)** *mal...*
  * 3. **O(n)**
  + 4. **O(n)**
+ 5. **O(n)**

Gesamtlaufzeit: $O(n) + (\log_2 n + 1) * O(n) + O(n)$

$= O(n) + O(n + n*\log_2 n) + O(n)$

$= O(n*\log_2 n)$

Zusammenfassung

- Komplexitätsanalyse von Algorithmen ist ein Werkzeug um festzustellen, um den Aufwand zur Lösung eines Problems zu bestimmen.

- Verschiedene Abstraktionen vom Konkreten Problem helfen dabei, die Analyse einfach und allgemein zu halten:

  - ein abstraktes Maschinenmodell
  - einziger Parameter: Länge der Eingabe
  - O-Notation und Einschränkung auf "*worst case*"-Komplexität

Jetzt:

Wir unterteilen die Klasse der entscheidbaren Probleme nach dem für ihre Lösung benötigten Zeitaufwand.

Fragen bis hier?

2. Klassifikation von Problemen anhand ihrer Zeitkomplexität

Was ist eine Zeitkomplexitätsklasse?

Definition:

Sei $t : \mathbb{N} \longrightarrow \mathbb{N}$ eine beliebige Funktion, dann ist die **Zeitkomplexitätsklasse** *TIME(t(n))* definiert durch:

TIME(t(n)) = {L | L ist eine Sprache, die von einer *O(t(n))*-Zeit-TM entschieden wird. }

Also:

- Definition basiert auf O-Notation
- *zur Erinnerung:*

  Eine "f(n)-Zeit-Turingmaschine" ist entsprechend unserer Definition eine deterministische TM.

Für unser Problem A...

Wir haben zuerst gezeigt:

$$A \in TIME(n^2)$$

Dann sogar:

$$A \in TIME(n \cdot log(n))$$

Anmerkung:

- *f(n) = O(g(n))* und *g(n) = O(h(n))* $\rightarrow$ *f(n) = O(h(n))*
- Jedes Problem das in *n·log(n)* Zeit gelöst werden kann, kann auch in *n²* Zeit gelöst werden.

Ein anderes Modell

Eine 2-Band-Turingmaschine kann das Problem in linearer Zeit entscheiden.

Was bedeutet das im Zusammenhang mit der Definition der Zeitkomplexitätsklasse?

Satz

Sei $t(n)$ eine Funktion mit $t(n) \geq n$, dann:

Für jede t(n)-Zeit-Mehrband-TM gibt es eine äquivalente $O(t^2(n))$-Zeit-Einband-Turingmaschine.

Idee für eine simulierende 1-Band-TM

Simulation der *k* Bänder der Mehrband-TM durch aufeinanderfolgende Abschnitte auf dem Band der 1-Band-Turingmaschine

Markierung der Kopfposition durch gesondertes Zeichen

Phasen der Simulation eines Schrittes der Mehrband-TM:

1. Phase:

ein Lauf über das komplette Band um Kopfpositionen und aktuell gelesene Zeichen zu bestimmen

2. Phase:

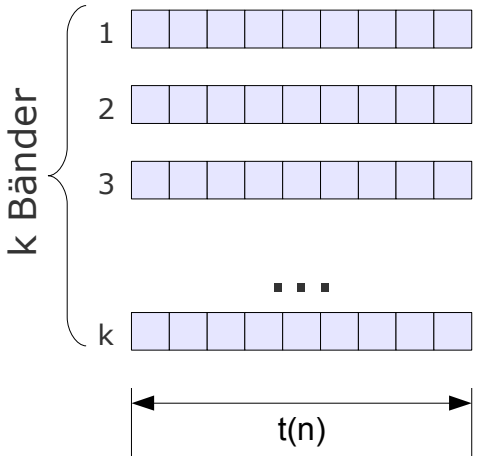
ein weiterer Durchlauf zur Aktualisierung von Kopfpositionen und ggf. Bandinhalten.

Bei Vergrößerung von Bandinhalten der simulierten *k*-Band-TM:

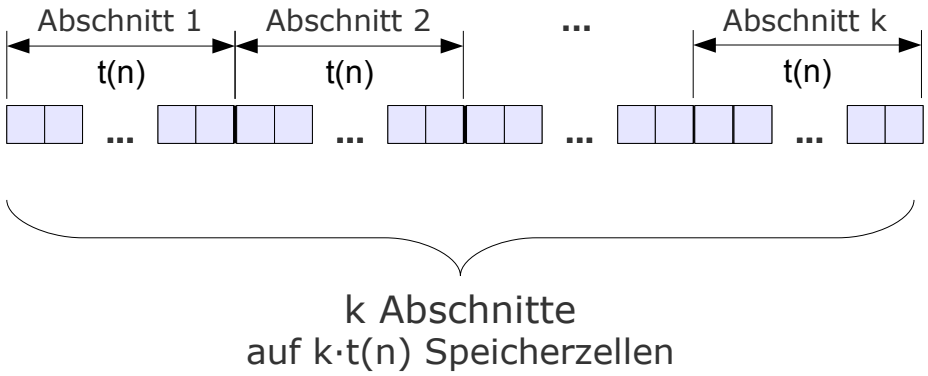
Verschieben des Bandinhaltes der simulierenden 1-Band-TM

...anschaulicher

k-Band-Turingmaschine



1-Band-Turingmaschine



Laufzeit der Simulation

1 Schritt der *k*-Band-Turingmaschine auf der 1-Band-TM

1. Phase: Kopfpositionen und gelesene Zeichen bestimmen

pro Abschnitt: $O(t(n))$ Schritte

für k Abschnitte: $k \cdot O(t(n)) =$ **$O(t(n))$ Schritte**

2. Phase: Aktualisieren von Kopfpositionen und Bandinhalten

pro Abschnitt: $O(1) + O(n) = O(n)$

für k Abschnitte: $k \cdot O(t(n)) =$ **$O(t(n))$ Schritte**

Insgesamt für 1 Schritt auf der 1-Band-TM: **$O(t(n))$ Schritte**

Laufzeit der Simulation

Wir wissen:

- gegebene *k*-Band-Turingmaschine hat Laufzeit *t(n)*
- Simulation eines der *t(n)* Schritte benötigt *O(t(n))* Zeit

Also sind um t(n) Schritte auf der 1-Band-Turingmaschine zu simulieren:

$$t(n) \cdot O(t(n)) = \mathbf{O(t^2(n))}\ \mathbf{Schritte}$$

Ein Fazit

1-Band- und Mehrband-Turingmaschine bezüglich Entscheidbarkeit äquivalent, d.h. sie entscheiden die gleiche Klasse von Sprachen.

Aber:

Beide Modelle sind bezüglich Zeitkomplexität nicht äquivalent.

$$O(t(n)) \nleftrightarrow O(t^2(n))$$

Die Laufzeit eines gewissen Lösungsverfahrens/Algorithmus ist abhängig vom zugrunde liegenden Berechnungsmodell!

Nichtdeterminismus?

Was bedeutet Nichtdeterminismus in diesem Zusammenhang?

Satz

Sei $t(n)$ eine Funktion mit $t(n) \geq n$, dann:

Für jede nichtdeterministische $t(n)$-Zeit-TM gibt es eine äquivalente deterministische $2^{O(t(n))}$-Zeit-Turingmaschine.

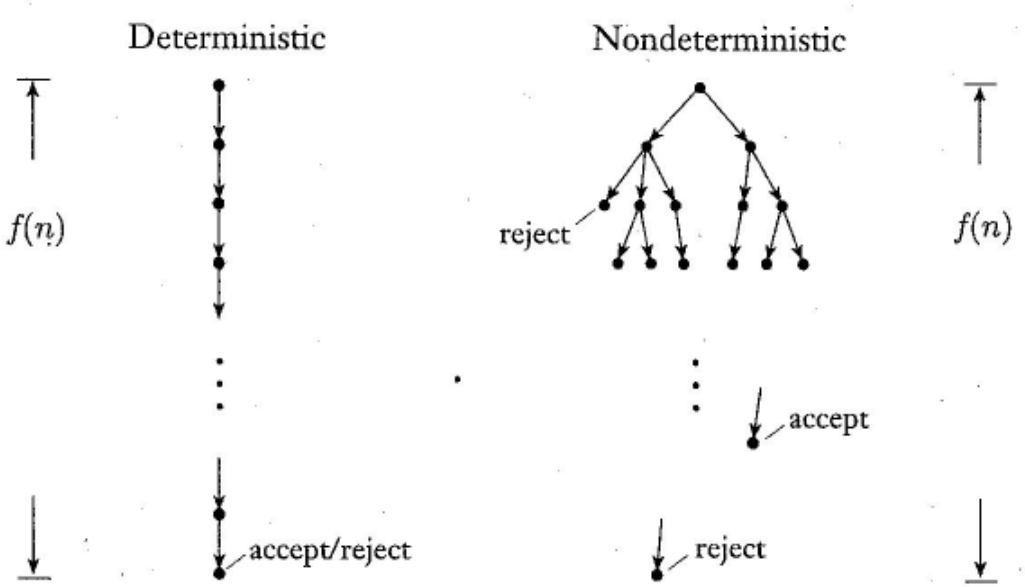
Also:

Die Simulation einer nichtdeterministischen Turingmaschine durch eine deterministische ist mit exponentiellem Aufwand verbunden!

Nichtdeterministische TM (NTM)

anschaulich:

Rechnung einer NTM im Vergleich zur deterministischen TM



Nichtdeterministische TM (NTM)

in Kürze:

Eine NTM **entscheidet** eine Sprache A, falls

… sie auf allen Zweigen ihrer Rechnung (nach endlicher Zeit) hält.

Die **Laufzeit** einer NTM

ist eine Funktion $f: \mathbb{N} \rightarrow \mathbb{N}$

wobei *f(n)* die die Maximalanzahl von Schritten ist, die *M* auf einem Zweig ihrer Rechnung bei einer Eingabe der Länge *n* macht.

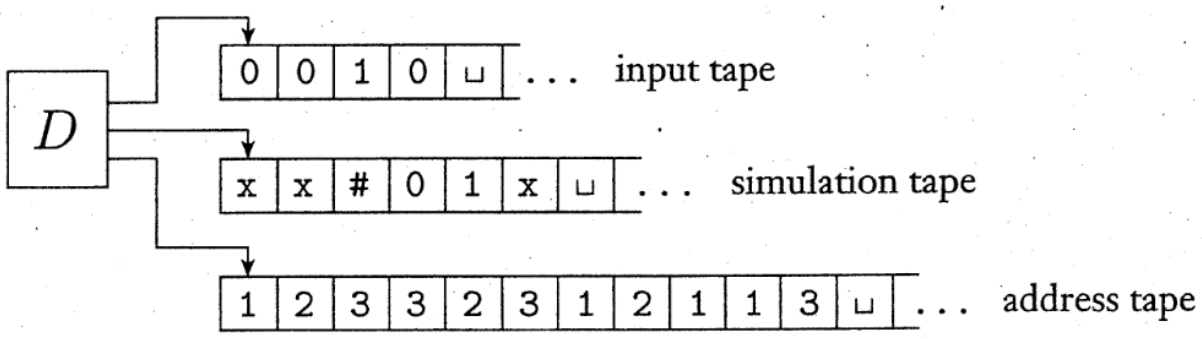
Die deterministische Simulation

Idee für die Konstruktion einer deterministische 3-Band-Turingmaschine D für eine NTM N:

Bänder:

1. Band – Eingabe von D
2. Band – Simulation
3. Band – der aktuell simulierte Pfad von N

anschaulich:



Die deterministische Simulation

Funktionsweise der 3-Band-TM D:

- Die Eingabe auf Band 1 wird auf das 2. Band kopiert.
- Alle Rechnungen von N werden Zweig für Zweig auf dem 2. Band simuliert.
- Inhalt des 3. Bandes gibt Pfad durch den Berechnungsbaum von N entsprechend dessen Übergangsfunktion vor:

  "1233..." heißt: *"Gehe (von der Wurzel ausgehend) zum 1. Kindknoten, dann zu dessem 2. Kind und zu dessem 3. …"*

- Nach Beendigung der Simulation eines Zweigs oder falls das "Adresswort" einen ungültigen Zustandsübergang vorgibt:

  … weiter mit dem lexikografisch nächsten "Adresswort"/Berechnungszweig

- Akzeptieren: sobald eine akzeptierende Konfiguration von N gefunden wurde
- Verwerfen: sonst.
- Adressraum beschränkt durch maximale Anzahl von Folgekonfigurationen eines Knotens in der Übergangsfunktion von N

Die Laufzeit der Simulation

Die Idee hinter der $2^{O(t(n))}$-Zeitbeschränkung:

- Bei Eingabelänge n hat jeder Zweig der Rechnung einer t(n)-Zeit-NTM N höchstens die Länge $t(n)$.
- Die Anzahl der Kindknoten (Folgezustände) eines Knotens (Zustands) im Berechnungsbaum von N ist durch eine Konstante b beschränkt.

  $\rightarrow$ b ist gegeben durch das Maximum der möglichen Optionen für Folgekonfigurationen in der Übergangsfunktion von N
- Für die Anzahl p von Blättern folgt:

  $p \leq b^{t(n)}$
- Für die Gesamtanzahl k von Knoten:

  $k < 2 \cdot p \rightarrow k = O(b^{t(n)})$
- Zusammen mit einer "worst case"-Laufzeit $t(n)$ von N:

  Gesamtlaufzeit von D: $O(t(n) \cdot b^{t(n)}) = \mathbf{2^{O(t(n))}}$

Zusammenfassung (bis hier)

- Unterteilung der Klasse der entscheidbaren Probleme durch Definition der Zeitkomplexitätsklasse
- durch O-Notation:
  - Klassifikation bietet gewisse Toleranz gegen Laufzeitdifferenzen
  - Teilmengenbeziehung $TIME(f(n)) \subset TIME(g(n))$, falls $f(n) = O(g(n))$
- **Aber:** unterschiedliche Laufzeiten auf unterschiedlichen Modellen
  - k-Band-TM definiert eine andere Zeitkomplexitätsklasse *TIME* als 1-Band-TM
  - Nichtdeterminismus führt bei Simulation auf einem deterministischen Modell zu *"enormen"* Laufzeiten

  (das einführende Beispiel macht die Größenordnung deutlich)

Fragen bis hier?

3. Die Komplexitätsklasse P

Was ist P?

Definition:

P ist die Klasse der Sprachen die durch eine deterministische Turingmaschine in polynomieller Zeit entscheidbar sind:

$$P = \bigcup_k TIME(n^k)$$

Also:

- Zur "weiten Fassung" von *TIME(f(n))* durch O-Notation tolerieren wir nun auch polynomielle Differenzen.

Beispiel: 2 Probleme *Q* und *R* mit $Q \in TIME(n \cdot log(n)))$ und $R \in TIME(n^{10000})$

→ sowohl *Q* als auch *R* sind in der Klasse *P*

Klasse der "effizient" lösbaren Probleme

P wird auch als Klasse der *"effizient"* lösbaren Probleme bezeichnet.

Was heißt "effizient"?

- keine klare mathematische Definition (möglich)
- verschiedene Auffassungen

Wie sinnvoll ist die Definition von P?

- Zitat (Wegener: Kompendium Theoretische Informatik):

  "… Es gibt keine 'bessere' Klasse als P zur Charakterisierung effizient lösbarer Probleme."

Vorteile der so gewählten Definition

1. Wir sind unabhängig von konkreten Maschinen und Berechnungsmodellen
   - Zur 1-Band-TM polynomiell äquivalente Berechnungsmodelle
   - z.B.: Mehrband-TM, Registermaschine

2. Polynomielle Algorithmen sind für die meisten Zwecke *brauchbar*, exponentielle in den meisten Fällen nicht:
   - unser einführendes Beispiel
   - Aber: Keine Gesetzmäßigkeit!

Welche Probleme sind in P?

… alle Algorithmen die wir bisher in ALP & Co. kennengelernt haben.

Einige Beispielprobleme in P und ihre Lösungsverfahren:

- Wortproblem für reguläre Sprachen

  Matching-Verfahren z.B. in der Java-Standardbibliothek oder den Suchfunktionen vieler Office-Anwendungen

- Wortproblem kontextfreier Sprachen

  CYK-Algorithmus

- RELPRIME (*"Sind zwei Zahlen relativ prim?"*)

  z.B. unter Verwendung des Euklidischen Algorithmus

- PRIME (*"Ist eine gegebene Zahl Primzahl?"*)

  AKS-Primzahltest (2002)

  PATH (*"Existiert Weg zwischen zwei Knoten eines gerichteten Graphen"*)

  z.B. durch Breiten-/Tiefensuche

Welche Probleme sind nicht in P?

Von Problemen, für die nur Lösungsverfahren mit deterministisch exponentiellen Zeitaufwand bekannt sind, wissen wir zunächst nur, dass sie in $TIME(k^n)$ sind (für eine Konstante k).

Die Definition von $TIME(f(n))$ ist *"positiv"*.

- Sie gibt keinen Ausschlussgrund für die Zugehörigkeit zu einer Zeitkomplexitätsklasse vor.
- Das schließt nicht aus, dass gewisse Probleme aufgrund ihrer Beschaffenheit bzgl. Zeitkomplexität nach unten beschränkt sind.

  Beispiel: vergleichsbasiertes Sortieren

Viele Fragen nach (un-)echten Teilmengenbeziehungen zwischen Zeitkomplexitätsklassen (z.B. P-NP-Problem) sind bis jetzt offen verblieben.

Zusammenfassend

Nicht jedes entscheidbare Problem ist auch "effizient" lösbar.

Abstraktion hilft bei einer modell-/maschinenunabhängigen Klassifizierung von Problemen nach ihrer "Schwere".

Im konkreten Fall ist die Analyse eines Algorithmus ein nützliches Mittel um eine Aussage über dessen Effizienz machen zu können und i.w.S. um die "Schwere" des vorliegenden Problems einzuschätzen.

… über den Tellerrand

P-Vollständigkeit

P-Vollständigkeit (*informal*)

Ein Problem A in P heißt "P-vollständig", falls sich jedes Problem in P mit einem beschränkten Resourcenbedarf auf A reduzieren lässt.

Logarithmisch platzbeschränkte Reduktion

- in GTI: "A auf B reduzierbar" ⟷ A ist "höchstens so schwer" wie B.
- logarithmisch platzbeschränkt:

  → Reduktionsfunktion muss auf logarithmisch beschränktem Platz berechenbar sein.

  → ein Mittel um P-Vollständigkeit zu zeigen

CVP ist P-Vollständig

- Circuit Value Problem
- "Ist die Ausgabe eines gegebenen Schaltkreises bei bestimmten Eingaben *1*?"

Geht es schneller als *effizient*?

Eine Situation ähnlich der, ob es "effiziente" deterministische Algorithmen für Probleme in NP gibt:

- parallel "effizient" lösbare Probleme → Klasse NC $\subseteq$ P (Nick's Class)
- eine weitere relevante Ressource hier: Anzahl der Prozessoren
- Wie bei P und NP die Frage nach der Art der Teilmengenbeziehung:
  → Es gibt effiziente parallele Algorithmen für Probleme in P
  → Jedoch wurde bisher für keines der P-vollständigen Probleme ein solcher Algorithmus gefunden.

Falls es einen effizienten parallelen Algorithmus für eines der P-vollständigen Probleme gibt:

- Es gibt solche "hochparallelen" Algorithmen für alle Probleme, die sequentiell effektiv lösbar sind (also denen in P).

… zum Weiterlesen

R. Greenlaw, H. J. Hoover, S. Miyano, W. L. Ruzzo, S. Shiraishi and T. Shoudai

The Parallel Computation Project: Volumes I-III (2000)

digital verfügbar unter:

http://www.cs.armstrong.edu/greenlaw/research/PARALLEL/

Gibt es noch offene Fragen?

Danke!

Präsentation & Handout ab morgen als PDF verfügbar unter:

http://page.mi.fu-berlin.de/sstugk/proseminar