Vorlesung 2: Karatsuba, Asymptotische Notation, Induktion, Star Suche

1 Karatsuba's Algorithmus

> *Karatsuba's Algorithmus führt wesentlich weniger elementare Operationen als Schulmethode aus*
>
> *auf diese Weise können Computer grosse Zahlen viel schneller Multiplizieren*
>
> *wichtig für Kryptographie und genaue numerische Simulationen*
>
> *z.B. fuer Zahlen mit 16000 Stellen ist Karatsuba mehr als ein Faktor 10 schneller*

1.1 Fall n=2

	a_1	a_0	×	b_1	b_0	
	6	**5**		**2**	**1**	
					5	$a_0 \cdot b_0$
		1	2			$a_1 \cdot b_1$
				-1		$-(a_1-a_0)\cdot (b_1-b_0)$
+			1	7		$a_1 \cdot b_1 + a_0 \cdot b_0$
		1	3	6	5	

Korrektheit: $a_1 \cdot b_1 + a_0 \cdot b_0 - (a_1-a_0)\cdot (b_1-b_0) = a_1 \cdot b_0 + a_0 \cdot b_1$

Resultat des Algorithmus: $a_1 \cdot b_1 \cdot 10^2 + (a_1 \cdot b_0 + a_0 \cdot b_1)\cdot 10 + a_0 \cdot b_0 = (a_1 \cdot 10 + a_0) \cdot (b_1 \cdot 10 + b_0) = a \cdot b$

> Ist dieser Algorithmus denn besser als die Schulmethode?

Scheinbar komplizierter: müssen subtrahieren

aber nur 3 MULT Operationen: $a_0 \cdot b_0$, $a_1 \cdot b_1$, $(a_1-a_0)\cdot (b_1-b_0)$

anstatt 4 MULT Operationen nach Schulmethode: $a_0 \cdot b_0$, $a_1 \cdot b_1$, $a_1 \cdot a_0$, $a_0 \cdot b_1$

Subtraktion: ähnlich wie Addition mit zusätzlicher elementaren Operation (Subtraktion von Ziffern) — siehe Übungsblatt

1.2 Fall n=4

> Können wir dieselbe Idee wie für n=2 verwenden?

	a_1	a_0	×	b_1	b_0	
	87	**65**		**43**	**21**	
				13	65	$a_0 \cdot b_0$
		37	41			$a_1 \cdot b_1$
+			46	22		$a_1 \cdot b_1 + a_0 \cdot b_0 - (a_1 - a_0) \cdot (b_1 - b_0)$
		37	87	35	65	

> *hier ist $(a_1-a_0)\cdot(b_1-b_0) = 22 \cdot 22 = 484$*

Korrektheit: wie für n=2

> Wie viele MULT Operationen macht dieser Algorithmus?

Berechne Teilprodukte $a_0 \cdot b_0$, $a_1 \cdot b_1$, $(a_1-a_0)\cdot (b_1-b_0)$ mit je 3 MULT Operationen (Verfahren für n=2)

Anzahl MULT Operationen: $3 \cdot 3 = 9$

1.3 Allgemeines $n=2^k$ (Zweierpotenz):

> *- zerlege $a=a_1 \cdot 10^{n/2} + a_0$ und $b = b_1 \cdot 10^{n/2} + b_0$ // n/2-stellige Zahlen a_1, a_0, b_1, b_0*
> *- berechne $a_0 \cdot b_0$, $a_1 \cdot b_1$, $(a_1-a_0)\cdot(b_1-b_0)$ rekursiv*
> *- **RETURN** $a_1 \cdot b_1 \cdot 10^n + (a_0 \cdot b_0 + a_1 \cdot b_1 - (a_1-a_0)\cdot(b_1-b_0)) \cdot 10^{n/2} + a_0 \cdot b_0$*

rekursiv: wende dasselbe Verfahren auf kleinere Instanzen des Problems an

Basisfall: n=1, MULT(a_0, b_0)

allgemeines Prinzip: DIVIDE & CONQUER

> *DIVIDE: im zweiter Schritt lösen wir Teile des Problems unabhängig voneinander CONQUER: im dritten Schritt kombinieren wir die Teillösungen zur Gesamtlösung*

Anzahl MULT Operationen: $T_{MULT}(n)$

es gilt: $T_{MULT}(2^k) = 3 \cdot T_{MULT}(2^{k-1})$ sowie $T_{MULT}(1)=1$ (Basisfall)

wiederholtes Einsetzen:

$T_{MULT}(2^k)$
$= 3 \cdot T_{MULT}(2^{k-1})$
$= 3 \cdot 3 \cdot T_{MULT}(2^{k-2})$
$\dots$
$= 3^i \cdot T_{MULT}(2^{k-i})$
$\dots$
$= 3^k \cdot T_{MULT}(2^0) = 3^k$

da $n=2^k$ ist $3^k = (2^{\log_2(3)})^k = (2^k)^{\log_2(3)} = n^{\log_2(3)} \leq n^{1.585}$

> *genauer $\log_2(3) \approx 1.58496250072$*

Schulmethode macht mehr als $n^{0.4}$-fach mehr Multiplikationen

> *je grösser n desto grösser ist die Verbesserung*
>
> *wir sagen $n^{1.585}$ hat strikt kleineres asymptotischen Wachstum als n^2*

2 Algorithmen vergleichen und asymptotische Notation

> *Oft sind viele Algorithmen möglich für ein Problem*
>
> *Wie können wir zwei Algorithmen vergleichen?*

Praxis: Laufzeit auf Computer

> *die Laufzeit ist leicht messbar, aber schwierig vorhersehbar*

Theorie: Anzahl elementarer Operationen

> *Hier ist eine mathematische Analyse möglich!*

> *Stimmen Theorie und Praxis immer überein?*

Gründe für Diskrepanz:

- welche elementaren Operationen?
- welche Computerarchitektur?

- welche Implementation? (Python, Java, C++, ...)

> *genaue Kosten von elementaren Operationen hängen stark von der Architektur des Rechners und anderen Faktoren ab*

Idee:

- Fokus auf asymptotisches Wachstum
- ignoriere konstante Faktoren

Formal: asymptotische Notation

Definition: sei $f: N \to \mathbb{R}^+$, dann

$$O(f) = \left\{ g: N \to \mathbb{R}^+ \mid \exists C > 0. \forall n \in N. \, g(n) \leq C \cdot f(n) \right\}$$

meist: N={1,2,3,...}

> *In Worten, O(f) ist die Menge aller Funktionen g mit folgender Eigenschaft: es gibt eine positive Konstante C so dass die Ungleichung g(n)≤ C · f(n) für alle n gilt*

intuitiv: alle Funktionen g, die durch f beschränkt bis auf einen konstante Faktor sind

> *hier sind unterschiedliche Definitionen möglich*
>
> *sie stimmen aber bis auf Randfälle miteinander überein*

> *wichtige Eigenschaften anhand von Beispielen*

konstante Faktoren ignorieren: O(n) = O(1000· n) = O(n/1000)

vereinfachen: O(10· n^2 + 100· n + 1000) = O(n^2)

asymptotisches Wachstum zählt: O(1) ≠ O(n) ≠ O(n^2)

> *Nützliche Schreibweise*

Konvention: wir schreiben g ≤ O(f) wenn g∈ O(f) gilt

Beispiel: 10 · n^2 + 100 · n + 100 ≤ O(n^2)

Satz: O(f)=O(g) genau dann wenn g ≤ O(f) und f≤ O(g)

3 Vollstaendige Induktion

3.1 Beispiel: Rekurrenzungleichung

T(n) ≤ 3· T(n/2) + 7 · n T(1) = 1

> *Karatsuba's Anzahl der elementaren Operationen erfüllt diese Rekurrenzungleichung*
>
> - *2 Additionen von n/2 stelligen Zahlen: A = a_1 + (- a_0) und B= b_1 + (- b_0)*
> - *2 Additionen von n stelligen Zahlen: Z = $a_1 \cdot b_1 + a_0 \cdot b_0 + A \cdot B$*
> - *2 Additionen von 2n-stelligen Zahlen $a_1 b_1 \cdot 10^k + Z \cdot 10^{k/2} + a_0 b_0$*
> - *insgesamt 7· n ADD Operationen (im rekursiven Fall)*
> - *im Basisfall n=1 brauchen wir nur eine MULT Operationen*

> *Wir können wiederholtes Einsetzen für Rekurrenzungleichung verwenden*
>
> - *gut um Idee zu bekommen*
> - *aber oft umständlich für sauberen Beweis*

> *mit vollständiger Induktion können wir leicht einen sauberen Beweis geben*

zu beweisen: $T(n) \leq 15 \cdot n^{\log_2(3)} - 14 \cdot n$ für alle Zweierpotenzen $n=2^k$

also: $T(2^k) \leq 15 \cdot 3^k - 14 \cdot 2^k$ für alle $k \geq 0$

Induktionsanfang (k=0): $15 \cdot 3^k - 14 \cdot 2^k = 15 - 14 = 1 = T(1)$

Induktionsschritt (k→ k+1):

> *$T(2^{k+1})$*
> *$\leq 3 \cdot T(2^k) + 7 \cdot 2^{k+1}$ (Rekurrenz für k+1)*
> *$\leq 3 \cdot (15 \cdot 3^k - 14 \cdot 2^k) + 7 \cdot 2^{k+1}$ (Induktionsannahme für k)*
> *$= 15 \cdot 3^{k+1} - 14 \cdot 2^{k+1} - 14 \cdot 2^k + 7 \cdot 2^{k+1}$ (Umformen)*
> *$= 15 \cdot 3^{k+1} - 14 \cdot 2^{k+1}$*

> *Die Faktoren 15 und 14 sind hier sorgfältig gewählt*
>
> *Um eine gute Wahl zu treffen, können wir Variablen für diese Faktoren einführen und Bedingungen für die Variablen finden, indem wir den Induktionsbeweis durchführen*

3.2 Allgemeine Form

Aussage A(k) über eine natürliche Zahl $k \in N$

> *wir wollen beweisen dass diese Aussage für alle natürlichen Zahlen k gilt*
>
> *dazu beweisen wir zunaechst dass A(1) gilt*
>
> *und dann beweisen fuer alle $k \geq 1$:*
>
> *die Aussage A(k+1) gilt unter der Annahme dass A(k) gilt*
>
> *d.h. wir muessen die Aussage A(k+1) beweisen aber duerfen im Beweis annehmen dass die Aussage A(k) gilt*

4 Star Suche

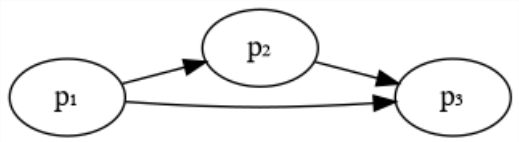
Problem: suche "Star" unter n Personen $p_1, ..., p_n$ $(n \geq 2)$

Definition: Person ist "Star" falls

- jeder sie kennt
- sie keinen kennt

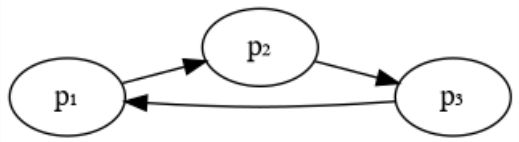
> *Stellen sie sich vor Donald Trump kommt zur Vorlesung*

Beispiele (n=3):



> Wer ist der Star hier?

Star: p_3



> Wer ist der Star hier?

kein Star

> Kann es mehr als einen Star geben?

immer: ≤ 1 Star

> *Wir wollen einen Algorithmus für dieses Problem entwickeln*
>
> *Wir legen dazu folgende Frage als elementare Operation fest:*

elementare Operation: i ≠ j

KNOWS(i,j) = $\begin{cases} 1 & p_i \text{ kennt } p_j \\ 0 & \text{anderenfalls} \end{cases}$

naive Suche: alle n· (n-1) Fragen

rekursiver Ansatz (n>2)

1. Suche Star p_s unter p_1 , ..., p_{n-1} rekursiv
2. Teste ob p_s Star für p_1 , ... , p_n: 2 Fragen
3. Falls nicht, teste ob p_n Star für p_1, ..., p_n: 2· (n-1) Fragen

guter Fall:

$2 + 2 + \cdots +

schlechter Fall:

$$2 \cdot (n-1) + 2 \cdot (n-2) + \cdots + \underbrace{2 \cdot 2}_{p_1,p_2,p_3} + \underbrace{2}_{p_1,p_2} = n^2$$

geht es besser?

Stelle sicher dass p_n kein Star: 1 Frage

> *Wie geht das?*

> *IF KNOWS(n-1,n)=0 {*
> *// p_n kann kein Star sein*
> *} ELSE {*

```
  // p_{n-1} kann kein Star sein
  vertausche p_{n-1} und p_n
}
```