5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

Was messen wir?

- Speicherverbrauch
- Laufzeit
- Anzahl Vergleiche (z.B. Sortieralgorithmen)
- Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- Festplattenzugriffe
- Programmgröße
- Energieverbrauch
- ...

5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

Was messen wir?

- Speicherverbrauch
- Laufzeit
- Anzahl Vergleiche (z.B. Sortieralgorithmen)
- Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- Festplattenzugriffe
- Programmgröße
- Energieverbrauch
- ...

5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

Was messen wir?

- Speicherverbrauch
- Laufzeit
- Anzahl Vergleiche (z.B. Sortieralgorithmen)
- Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- Festplattenzugriffe
- Programmgröße
- Energieverbrauch
- ...

5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

Was messen wir?

- Speicherverbrauch
- Laufzeit
- Anzahl Vergleiche (z.B. Sortieralgorithmen)
- Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- Festplattenzugriffe
- Programmgröße
- Energieverbrauch
- ...

5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

Was messen wir?

- Speicherverbrauch
- Laufzeit
- Anzahl Vergleiche (z.B. Sortieralgorithmen)
- Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- Festplattenzugriffe
- Programmgröße
- Energieverbrauch
- ...

5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

Was messen wir?

- Speicherverbrauch
- Laufzeit
- Anzahl Vergleiche (z.B. Sortieralgorithmen)
- Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- Festplattenzugriffe
- Programmgröße
- Energieverbrauch
- ...

5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

Was messen wir?

- Speicherverbrauch
- Laufzeit
- Anzahl Vergleiche (z.B. Sortieralgorithmen)
- Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- Festplattenzugriffe
- Programmgröße
- Energieverbrauch
- ...

5 Effizienz von Algorithmen

Was heißt ein Algorithmus ist effizient?

Was messen wir?

- Speicherverbrauch
- Laufzeit
- Anzahl Vergleiche (z.B. Sortieralgorithmen)
- Anzahl an Multiplikationen (wissenschaftliches Rechnen)
- Festplattenzugriffe
- Programmgröße
- Energieverbrauch
- ...

5 Effizienz von Algorithmen

Wie messen wir?

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.

- Theoretische Analyse in einem Rechenmodell.
  - Gibt [illegible] wie z.B. „dieser Algorithmus läuft immer in Zeit $\mathcal{O}(n^2)$".
  - Üblicherweise wird der [illegible] betrachtet.
  - Man kann auch untere Schranken erhalten: „Jedes vergleichsbasierte Sortierverfahren benötigt im worst case mindestens $\Omega(n \log n)$ Vergleiche".

5 Effizienz von Algorithmen

Wie messen wir?

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.

- Theoretische Analyse in einem Rechenmodell.

5 Effizienz von Algorithmen

Wie messen wir?

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.
- Theoretische Analyse in einem Rechenmodell.

5 Effizienz von Algorithmen

Wie messen wir?

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.
- Theoretische Analyse in einem Rechenmodell.

5 Effizienz von Algorithmen

Wie messen wir?

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.
- Theoretische Analyse in einem Rechenmodell.

5 Effizienz von Algorithmen

Wie messen wir?

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.

- Theoretische Analyse in einem Rechenmodell.
  - Gibt asymptotische Garantien wie z.B. „dieser Algorithmus läuft immer in Zeit $\mathcal{O}(n^2)$".
  - Üblicherweise wird der worst case betrachtet.
  - Man kann auch untere Schranken erhalten: „jedes vergleichsbasierte Sortierverfahren benötigt im worst case mindestens $\Omega(n \log n)$ Vergleiche".

5 Effizienz von Algorithmen

Wie messen wir?

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.

- Theoretische Analyse in einem Rechenmodell.
  - Gibt asymptotische Garantien wie z.B. „dieser Algorithmus läuft immer in Zeit $\mathcal{O}(n^2)$“.
  - Üblicherweise wird der worst case betrachtet.
  - Man kann auch untere Schranken erhalten: „jedes vergleichsbasierte Sortierverfahren benötigt im worst case mindestens $\Omega(n \log n)$ Vergleiche“.

5 Effizienz von Algorithmen

Wie messen wir?

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.

- Theoretische Analyse in einem Rechenmodell.
  - Gibt asymptotische Garantien wie z.B. „dieser Algorithmus läuft immer in Zeit $\mathcal{O}(n^2)$".
  - Üblicherweise wird der worst case betrachtet.
  - Man kann auch untere Schranken erhalten: „jedes vergleichsbasierte Sortierverfahren benötigt im worst case mindestens $\Omega(n \log n)$ Vergleiche".

5 Effizienz von Algorithmen

Wie messen wir?

- Implementieren und Testen auf repräsentativen Eingaben.
  - Welche Eingaben?
  - Kann sehr aufwendig sein.
  - Präzise Resultate wenn sorgfältig durchgeführt.
  - Resultate gelten aber nur für spezifische Hardware, und spezifische Eingaben.

- Theoretische Analyse in einem Rechenmodell.
  - Gibt asymptotische Garantien wie z.B. „dieser Algorithmus läuft immer in Zeit $\mathcal{O}(n^2)$“.
  - Üblicherweise wird der worst case betrachtet.
  - Man kann auch untere Schranken erhalten: „jedes vergleichsbasierte Sortierverfahren benötigt im worst case mindestens $\Omega(n \log n)$ Vergleiche“.

5 Effizienz von Algorithmen

Eingabelänge

Die theoretischen Schranken werden als Funktion $f : \mathbb{N} \to \mathbb{N}$ von der Eingabelänge auf die Laufzeit (oder Speicherverbrauch, Energieverbrauch etc.) angegeben.

5 Effizienz von Algorithmen

Eingabelänge

Die theoretischen Schranken werden als Funktion $f : \mathbb{N} \to \mathbb{N}$ von der Eingabelänge auf die Laufzeit (oder Speicherverbrauch, Energieverbrauch etc.) angegeben.

Die Eingabelänge ist z.B.

- die Größe der Eingabe (Anzahl an Bits)
- die Anzahl der Argumente

Angenommen n Zahlen aus dem Bereich $\{1, \dots, N\}$ sollen sortiert werden. Wir sagen üblicherweise, dass die Eingabelänge n ist, anstatt z.B. $n \log N$, was der Anzahl an Bits entsprechen würde.

5 Effizienz von Algorithmen

Eingabelänge

Die theoretischen Schranken werden als Funktion $f : \mathbb{N} \to \mathbb{N}$ von der Eingabelänge auf die Laufzeit (oder Speicherverbrauch, Energieverbrauch etc.) angegeben.

Die Eingabelänge ist z.B.

- die Größe der Eingabe (Anzahl an bits)
- die Anzahl der Argumente

Example 1

Angenommen n Zahlen aus dem Bereich $\{1, \ldots, N\}$ sollen sortiert werden. Wir sagen üblicherweise, dass die Eingabelänge n ist, anstatt z.B. $n \log N$, was der Anzahl an Bits entsprechen würde.

5 Effizienz von Algorithmen

Eingabelänge

Die theoretischen Schranken werden als Funktion $f : \mathbb{N} \to \mathbb{N}$ von der Eingabelänge auf die Laufzeit (oder Speicherverbrauch, Energieverbrauch etc.) angegeben.

Die Eingabelänge ist z.B.

- die Größe der Eingabe (Anzahl an bits)
- die Anzahl der Argumente

Example 1

Angenommen n Zahlen aus dem Bereich $\{1, \dots, N\}$ sollen sortiert werden. Wir sagen üblicherweise, dass die Eingabelänge n ist, anstatt z.B. $n \log N$, was der Anzahl an Bits entsprechen würde.

5 Effizienz von Algorithmen

Eingabelänge

Die theoretischen Schranken werden als Funktion $f : \mathbb{N} \to \mathbb{N}$ von der Eingabelänge auf die Laufzeit (oder Speicherverbrauch, Energieverbrauch etc.) angegeben.

Die Eingabelänge ist z.B.

- die Größe der Eingabe (Anzahl an bits)
- die Anzahl der Argumente

Example 1

Angenommen n Zahlen aus dem Bereich $\{1, \ldots, N\}$ sollen sortiert werden. Wir sagen üblicherweise, dass die Eingabelänge n ist, anstatt z.B. $n \log N$, was der Anzahl an Bits entsprechen würde.

Rechenmodell

Wie messen wir

1. Berechne die Laufzeit in einem idealisierten Rechenmodell (z.B. Random Access Machine (RAM))
2. Berechne Anzahl von Basisoperationen wie z.B. Anzahl an Vergleichen, Multiplikationen, Festplattenzugriffen etc.

Rechenmodell

Wie messen wir

1. Berechne die Laufzeit in einem idealisierten Rechenmodell (z.B. Random Access Machine (RAM))
2. Berechne Anzahl von Basisoperationen wie z.B. Anzahl an Vergleichen, Multiplikationen, Festplattenzugriffen etc.

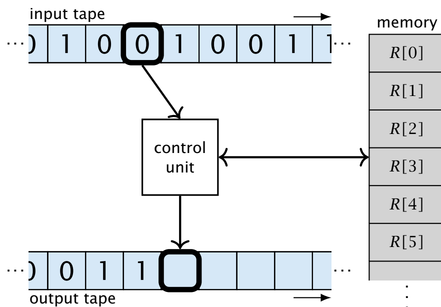
Rechenmodell

Wie messen wir

1. Berechne die Laufzeit in einem idealisierten Rechenmodell (z.B. Random Access Machine (RAM))
2. Berechne Anzahl von Basisoperationen wie z.B. Anzahl an Vergleichen, Multiplikationen, Festplattenzugriffen etc.

Random Access Machine (RAM)

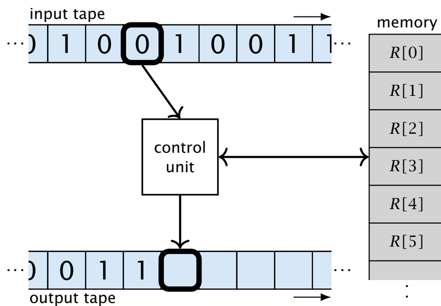
- Ein- und Ausgabeband (Folge von Einsen und Nullen; unbeschränkte Länge).
- Speicher: unendlich viele Register $R[0], R[1], R[2], \ldots$.
- Register können beliebige Integer speichern.
- Indirekte Adressierung.



Ein- und Ausgabeband sind gerichtet und ein Lese- oder Schreibzugriff bewegt das entsprechend Band zur nächsten Position.

Random Access Machine (RAM)

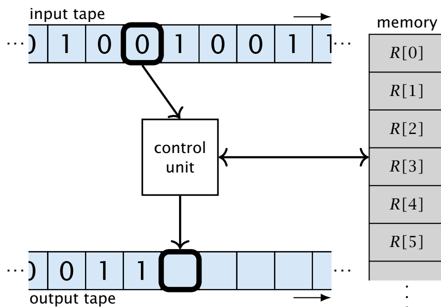
- Ein- und Ausgabeband (Folge von Einsen und Nullen; unbeschränkte Länge).
- Speicher: unendlich viele Register $R[0], R[1], R[2], \ldots$.
- Register können beliebige Integer speichern.
- Indirekte Adressierung.



Ein- und Ausgabeband sind gerichtet und ein Lese- oder Schreibzugriff bewegt das entsprechend Band zur nächsten Position.

Random Access Machine (RAM)

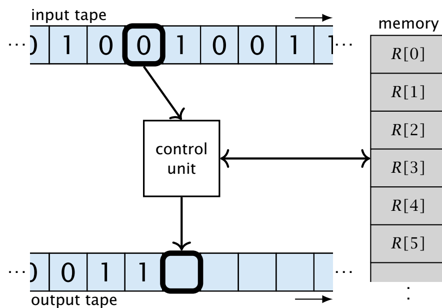
- Ein- und Ausgabeband (Folge von Einsen und Nullen; unbeschränkte Länge).
- Speicher: unendlich viele Register $R[0], R[1], R[2], \ldots$.
- Register können beliebige Integer speichern.
- Indirekte Adressierung.



Ein- und Ausgabeband sind gerichtet und ein Lese- oder Schreibzugriff bewegt das entsprechend Band zur nächsten Position.

Random Access Machine (RAM)

- Ein- und Ausgabeband (Folge von Einsen und Nullen; unbeschränkte Länge).
- Speicher: unendlich viele Register $R[0], R[1], R[2], \ldots$.
- Register können beliebige Integer speichern.
- Indirekte Adressierung.



Ein- und Ausgabeband sind gerichtet und ein Lese- oder Schreibzugriff bewegt das entsprechend Band zur nächsten Position.

Random Access Machine (RAM)

Operationen

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - READ i
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
- Registertransfers
- **indirekte** Adressierung

Random Access Machine (RAM)

Operationen

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - READ i
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - WRITE i
- Registertransfers
  - $R[j] := R[i]$
  - $R[j] := 4$
- **indirekte** Adressierung
  - $R[j] := R[R[i]]$
    lädt den Inhalt des $R[i]$-ten Registers in das j-te Register
  - $R[R[i]] := R[j]$
    lädt den Inhalt des j-ten Registers in das $R[i]$-te Register

Operationen

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - READ i
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)

Random Access Machine (RAM)

Operationen

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - `READ` i
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - `WRITE` i

Random Access Machine (RAM)

Operationen

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - READ i
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - WRITE i
- Registertransfers
  - $R[j]$:= $R[i]$
  - $R[j]$:= 4
- **indirekte** Adressierung
  - $R[j]$:= $R[R[i]]$
    lädt den Inhalt des $R[i]$-ten Registes in das j-te Register.
  - $R[R[i]]$:= $R[j]$
    lädt den Inhalt des j-ten Registers in das $R[i]$-te Register

Random Access Machine (RAM)

Operationen

- Eingabeoperationen (input tape → $R[i]$)
  - `READ` i
- Ausgabeoperationen ($R[i]$ → output tape)
  - `WRITE` i
- Registertransfers
  - $R[j]$ `:=` $R[i]$
  - $R[j]$ `:=` 4
- **indirekte** Adressierung

Random Access Machine (RAM)

Operationen

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - `READ` i
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - `WRITE` i
- Registertransfers
  - $R[j]$ `:=` $R[i]$
  - $R[j]$ `:=` 4
- **indirekte** Adressierung
  - $R[j]$ `:=` $R[R[i]]$
    lädt den Inhalt des $R[i]$-ten Registes in das j-te Register.
  - $R[R[i]]$ `:=` $R[j]$
    lädt den Inhalt des j-ten Registers in das $R[i]$-te Register

Random Access Machine (RAM)

Operationen

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - `READ` i
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - `WRITE` i
- Registertransfers
  - $R[j]$ `:=` $R[i]$
  - $R[j]$ `:=` 4
- **indirekte** Adressierung
  - $R[j]$ `:=` $R[R[i]]$
    lädt den Inhalt des $R[i]$-ten Registes in das j-te Register.
  - $R[R[i]]$ `:=` $R[j]$
    lädt den Inhalt des j-ten Registers in das $R[i]$-te Register

Random Access Machine (RAM)

Operationen

- Eingabeoperationen (input tape → $R[i]$)
  - READ i
- Ausgabeoperationen ($R[i]$ → output tape)
  - WRITE i
- Registertransfers
  - $R[j]$:= $R[i]$
  - $R[j]$:= 4
- **indirekte** Adressierung
  - $R[j]$:= $R[R[i]]$
    lädt den Inhalt des $R[i]$-ten Registes in das j-te Register.
  - $R[R[i]]$:= $R[j]$
    lädt den Inhalt des j-ten Registers in das $R[i]$-te Register

Random Access Machine (RAM)

Operationen

- Eingabeoperationen (input tape $\rightarrow R[i]$)
  - `READ` i
- Ausgabeoperationen ($R[i] \rightarrow$ output tape)
  - `WRITE` i
- Registertransfers
  - $R[j]$ `:=` $R[i]$
  - $R[j]$ `:=` 4
- **indirekte** Adressierung
  - $R[j]$ `:=` $R[R[i]]$
    lädt den Inhalt des $R[i]$-ten Registes in das j-te Register.
  - $R[R[i]] := R[j]$
    lädt den Inhalt des j-ten Registers in das $R[i]$-te Register

Random Access Machine (RAM)

Operationen

- Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - jump x
    springe zur Position x im Programm
    setze Befehlszähler auf x
    der nächste Befehl wird aus Register $R[x]$ gelesen
  - jumpz x $R[i]$
    springe zu x falls $R[i] = 0$
    falls nicht wird der Befehlszähler um 1 erhöht
  - jumpi i
    springe zu $R[i]$ (indirekter Sprung);
- arithmetische Operationen: $+$, $-$, $\times$, $/$
  - $R[i] := R[j] + R[k]$;
    $R[i] := -R[k]$;

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

Random Access Machine (RAM)

Operationen

- Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - `jump` x
    springe zur Position x im Programm
    setze Befehlszähler auf x
    der nächste Befehl wird aus Register $R[x]$ gelesen
  - `jumpz` x $R[i]$
    springe zu x falls $R[i] = 0$
    falls nicht wird der Befehlszähler um 1 erhöht
  - `jumpi` i
    springe zu $R[i]$ (indirekter Sprung);
- arithmetische Operationen: $+$, $-$, $\times$, $/$
  - $R[i]$:= $R[j] + R[k]$;
    $R[i]$:= $-R[k]$;

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

Random Access Machine (RAM)

Operationen

- Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - `jump` x
    springe zur Position x im Programm
    setze Befehlszähler auf x
    der nächste Befehl wird aus Register $R[x]$ gelesen
  - `jumpz` x $R[i]$
    springe zu x falls $R[i] = 0$
    falls nicht wird der Befehlszähler um 1 erhöht
  - `jumpi` i
    springe zu $R[i]$ (indirekter Sprung);
- arithmetische Operationen: $+$, $-$, $\times$, $/$
  - $R[i] := R[j] + R[k];$
    $R[i] := -R[k];$

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

Random Access Machine (RAM)

Operationen

- Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - `jump` x
    springe zur Position x im Programm
    setze Befehlszähler auf x
    der nächste Befehl wird aus Register $R[x]$ gelesen
  - `jumpz` x $R[i]$
    springe zu x falls $R[i] = 0$
    falls nicht wird der Befehlszähler um 1 erhöht
  - `jumpi` i
    springe zu $R[i]$ (indirekter Sprung);
- arithmetische Operationen: $+$, $-$, $\times$, $/$
  - $R[i]$:= $R[j]$ + $R[k]$;
    $R[i]$:= -$R[k]$;

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

Random Access Machine (RAM)

Operationen

- Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - `jump` x
    springe zur Position x im Programm
    setze Befehlszähler auf x
    der nächste Befehl wird aus Register $R[x]$ gelesen
  - `jumpz` x $R[i]$
    springe zu x falls $R[i] = 0$
    falls nicht wird der Befehlszähler um 1 erhöht
  - `jumpi` i
    springe zu $R[i]$ (indirekter Sprung);
- arithmetische Operationen: $+$, $-$, $\times$, $/$
  - $R[i]$:= $R[j]$ + $R[k]$;
    $R[i]$:= -$R[k]$;

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

Random Access Machine (RAM)

Operationen

- Verzweigungen (inklusive Schleifen) abhängig von Vergleichen
  - `jump` x
    springe zur Position x im Programm
    setze Befehlszähler auf x
    der nächste Befehl wird aus Register $R[x]$ gelesen
  - `jumpz` x $R[i]$
    springe zu x falls $R[i] = 0$
    falls nicht wird der Befehlszähler um 1 erhöht
  - `jumpi` i
    springe zu $R[i]$ (indirekter Sprung);
- arithmetische Operationen: $+$, $-$, $\times$, $/$
  - $R[i]$ `:=` $R[j]$ `+` $R[k]$;
    $R[i]$ `:=` `-`$R[k]$;

Die Sprungbefehle sind sehr ähnlich zu den Sprungbefehlen in verschiedenen Assemblersprachen.

Rechenmodell

Man nimmt normalerweise an, dass jeder Befehl eine Zeiteinheit kostet.

Komplexitätsschranken

Es gibt unterschiedliche Komplexitätsschranken:

- best-case Komplexität:

  $$C_{bc}(n) := \min\{C(x) \mid |x| = n\}$$

  Normalerweise einfach zu analysieren; nicht sehr hilfreich

- worst-case Komplexität:

  $$C_{wc}(n) := \max\{C(x) \mid |x| = n\}$$

  Standard. Machmal zu pessimistisch.

- average case Komplexität:

  $$C_{avg}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

  Manchmal schwierig zu analysieren.

$C(x)$	Kosten für Eingabe x
$\lvert x \rvert$	Eingabelänge von x
I_n	Menge der Eingaben mit Länge n

Komplexitätsschranken

Es gibt unterschiedliche Komplexitätsschranken:

- best-case Komplexität:

$$C_{\mathrm{bc}}(n) := \min\{C(x) \mid |x| = n\}$$

  Normalerweise einfach zu analysieren; nicht sehr hilfreich

- worst-case Komplexität:

$$C_{\mathrm{wc}}(n) := \max\{C(x) \mid |x| = n\}$$

  Standard. Machmal zu pessimistisch.

- average case Komplexität:

$$C_{\mathrm{avg}}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

  Manchmal schwierig zu analysieren.

$C(x)$	Kosten für Eingabe x
$\|x\|$	Eingabelänge von x
I_n	Menge der Eingaben mit Länge n

Komplexitätsschranken

Es gibt unterschiedliche Komplexitätsschranken:

- best-case Komplexität:

  $$C_{\mathrm{bc}}(n) := \min\{C(x) \mid |x| = n\}$$

  Normalerweise einfach zu analysieren; nicht sehr hilfreich

- worst-case Komplexität:

  $$C_{\mathrm{wc}}(n) := \max\{C(x) \mid |x| = n\}$$

  Standard. Machmal zu pessimistisch.

- average case Komplexität:

  $$C_{\mathrm{avg}}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

  Manchmal schwierig zu analysieren.

$C(x)$	Kosten für Eingabe x
$\lvert x\rvert$	Eingabelänge von x
I_n	Menge der Eingaben mit Länge n

Komplexitätsschranken

Es gibt unterschiedliche Komplexitätsschranken:

- best-case Komplexität:

  $$C_{\mathrm{bc}}(n) := \min\{C(x) \mid |x| = n\}$$

  Normalerweise einfach zu analysieren; nicht sehr hilfreich

- worst-case Komplexität:

  $$C_{\mathrm{wc}}(n) := \max\{C(x) \mid |x| = n\}$$

  Standard. Machmal zu pessimistisch.

- average case Komplexität:

  $$C_{\mathrm{avg}}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

  Manchmal schwierig zu analysieren.

$C(x)$	Kosten für Eingabe x
$\lvert x\rvert$	Eingabelänge von x
I_n	Menge der Eingaben mit Länge n

Asymptotische Notation

Wir interessieren uns normalerweise nicht für exakte Laufzeiten, sondern für eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren und additive Terme ignoriert.

- Wir interessieren uns für Laufzeiten bei großen Werten von n. Konstante additive Terme sind dann unwichtig.
- Eine superexakte Analyse (e.g. das exakte Zählen der Operationen auf einer RAM) ist schwierig, und würde die Resultate nicht verbessern, da das Rechenmodell die Realität nicht so exakt abbildet.
- Ein linearer speed-up (z.B. um einen konstanten Faktor) läßt sich z.B. erreichen wenn man den Algorithmus auf einem schnelleren Rechner laufen läßt.
- Laufzeiten sollte man durch einfache Funktionen ausdrücken können.

Asymptotische Notation

Wir interessieren uns normalerweise nicht für exakte Laufzeiten, sondern für eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren und additive Terme ignoriert.

- Wir interessieren uns für Laufzeiten bei großen Werten von n. Konstante additive Terme sind dann unwichtig.
- Eine superexakte Analyse (e.g. das *exakte* Zählen der Operationen auf einer RAM) ist schwierig, und würde die Resultate nicht verbessern, da das Rechenmodell die Realität nicht so exakt abbildet.
- Ein linearer speed-up (z.B. um einen konstanten Faktor) läßt sich z.B. erreichen wenn man den Algorithmus auf einem schnelleren Rechner laufen läßt.
- Laufzeiten sollte man durch einfache Funktionen ausdrücken können.

Asymptotische Notation

Wir interessieren uns normalerweise nicht für exakte Laufzeiten, sondern für eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren und additive Terme ignoriert.

- Wir interessieren uns für Laufzeiten bei großen Werten von n. Konstante additive Terme sind dann unwichtig.
- Eine superexakte Analyse (e.g. das *exakte* Zählen der Operationen auf einer RAM) ist schwierig, und würde die Resultate nicht verbessern, da das Rechenmodell die Realität nicht so exakt abbildet.
- Ein linearer speed-up (z.B. um einen konstanten Faktor) läßt sich z.B. erreichen wenn man den Algorithmus auf einem schnelleren Rechner laufen läßt.
- Laufzeiten sollte man durch einfache Funktionen ausdrücken können.

Asymptotische Notation

Wir interessieren uns normalerweise nicht für exakte Laufzeiten, sondern für eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren und additive Terme ignoriert.

- Wir interessieren uns für Laufzeiten bei großen Werten von n. Konstante additive Terme sind dann unwichtig.
- Eine superexakte Analyse (e.g. das *exakte* Zählen der Operationen auf einer RAM) ist schwierig, und würde die Resultate nicht verbessern, da das Rechenmodell die Realität nicht so exakt abbildet.
- Ein linearer speed-up (z.B. um einen konstanten Faktor) läßt sich z.B. erreichen wenn man den Algorithmus auf einem schnelleren Rechner laufen läßt.
- Laufzeiten sollte man durch einfache Funktionen ausdrücken können.

Asymptotische Notation

Wir interessieren uns normalerweise nicht für exakte Laufzeiten, sondern für eine asymptotische Klassifikation der Laufzeit, die konstante Faktoren und additive Terme ignoriert.

- Wir interessieren uns für Laufzeiten bei großen Werten von n. Konstante additive Terme sind dann unwichtig.
- Eine superexakte Analyse (e.g. das *exakte* Zählen der Operationen auf einer RAM) ist schwierig, und würde die Resultate nicht verbessern, da das Rechenmodell die Realität nicht so exakt abbildet.
- Ein linearer speed-up (z.B. um einen konstanten Faktor) läßt sich z.B. erreichen wenn man den Algorithmus auf einem schnelleren Rechner laufen läßt.
- Laufzeiten sollte man durch einfache Funktionen ausdrücken können.

Asymptotische Notation

Formale Definition

Sei f eine Funktion von $\mathbb{N}$ nach $\mathbb{R}^+$.

- $\mathcal{O}(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht schneller als f wachsen)

Asymptotische Notation

Formale Definition

Sei f eine Funktion von $\mathbb{N}$ nach $\mathbb{R}^+$.

- $\mathcal{O}(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht schneller als f wachsen)
- $\Omega(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht langsamer als f wachsen)

Asymptotische Notation

Formale Definition

Sei f eine Funktion von $\mathbb{N}$ nach $\mathbb{R}^+$.

- $\mathcal{O}(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht schneller als f wachsen)
- $\Omega(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht langsamer als f wachsen)
- $\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$
  (Funktionen die asymptotisch gleiches Wachstum wie f haben)

Asymptotische Notation

Formale Definition

Sei f eine Funktion von $\mathbb{N}$ nach $\mathbb{R}^+$.

- $\mathcal{O}(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht schneller als f wachsen)
- $\Omega(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht langsamer als f wachsen)
- $\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$
  (Funktionen die asymptotisch gleiches Wachstum wie f haben)
- $o(f) = \{g \mid \forall c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (Funktionen die asymptotische langsamer als f wachsen)

Asymptotische Notation

Formale Definition

Sei f eine Funktion von $\mathbb{N}$ nach $\mathbb{R}^+$.

- $\mathcal{O}(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht schneller als f wachsen)
- $\Omega(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$
  (Funktionen die asymptotisch nicht langsamer als f wachsen)
- $\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$
  (Funktionen die asymptotisch gleiches Wachstum wie f haben)
- $o(f) = \{g \mid \forall c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (Funktionen die asymptotische langsamer als f wachsen)
- $\omega(f) = \{g \mid \forall c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$
  (Funktionen die asymptotisch schneller als f wachsen)

Asymptotische Notation

Äquivalente Definition mit Grenzwerten (gilt nur falls der jeweilige Grenzwert existiert). f und g seien Funktionen von $\mathbb{N}_0$ nach $\mathbb{R}_0^+$.

- $g \in \mathcal{O}(f)$: $\quad 0 \le \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$

Asymptotische Notation

Äquivalente Definition mit Grenzwerten (gilt nur falls der jeweilige Grenzwert existiert). f und g seien Funktionen von $\mathbb{N}_0$ nach $\mathbb{R}_0^+$.

- $g \in \mathcal{O}(f)$: $\quad 0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in \Omega(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} \leq \infty$

Asymptotische Notation

Äquivalente Definition mit Grenzwerten (gilt nur falls der jeweilige Grenzwert existiert). f und g seien Funktionen von $\mathbb{N}_0$ nach $\mathbb{R}_0^+$.

- $g \in \mathcal{O}(f)$: $\quad 0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in \Omega(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} \leq \infty$
- $g \in \Theta(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$

Asymptotische Notation

Äquivalente Definition mit Grenzwerten (gilt nur falls der jeweilige Grenzwert existiert). f und g seien Funktionen von $\mathbb{N}_0$ nach $\mathbb{R}_0^+$.

- $g \in \mathcal{O}(f)$: $\quad 0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in \Omega(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} \leq \infty$
- $g \in \Theta(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in o(f)$: $\quad \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

Asymptotische Notation

Äquivalente Definition mit Grenzwerten (gilt nur falls der jeweilige Grenzwert existiert). f und g seien Funktionen von $\mathbb{N}_0$ nach $\mathbb{R}_0^+$.

- $g \in \mathcal{O}(f)$: $\quad 0 \le \lim\limits_{n\to\infty} \dfrac{g(n)}{f(n)} < \infty$
- $g \in \Omega(f)$: $\quad 0 < \lim\limits_{n\to\infty} \dfrac{g(n)}{f(n)} \le \infty$
- $g \in \Theta(f)$: $\quad 0 < \lim\limits_{n\to\infty} \dfrac{g(n)}{f(n)} < \infty$
- $g \in o(f)$: $\quad \lim\limits_{n\to\infty} \dfrac{g(n)}{f(n)} = 0$
- $g \in \omega(f)$: $\quad \lim\limits_{n\to\infty} \dfrac{g(n)}{f(n)} = \infty$

Asymptotische Notation

Missbrauch dieser Notation

1. Man schreibt $f = \mathcal{O}(g)$, anstatt $f \in \mathcal{O}(g)$. Dies ist **keine** Gleichheit (wie kann eine Funktion das gleiche wie eine Funktionsmenge sein?).
2. Man schreibt $f(n) = \mathcal{O}(g(n))$, anstatt $f \in \mathcal{O}(g)$, with $f : \mathbb{N} \to \mathbb{R}^+, n \mapsto f(n)$, and $g : \mathbb{N} \to \mathbb{R}^+, n \mapsto g(n)$.
3. Man schreibt $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$, anstatt $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$.

Asymptotische Notation

Missbrauch dieser Notation

1. Man schreibt $f = \mathcal{O}(g)$, anstatt $f \in \mathcal{O}(g)$. Dies ist **keine** Gleichheit (wie kann eine Funktion das gleiche wie eine Funktionsmenge sein?).
2. Man schreibt $f(n) = \mathcal{O}(g(n))$, anstatt $f \in \mathcal{O}(g)$, with $f : \mathbb{N} \to \mathbb{R}^+, n \mapsto f(n)$, and $g : \mathbb{N} \to \mathbb{R}^+, n \mapsto g(n)$.
3. Man schreibt $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$, anstatt $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$.

Asymptotische Notation

Missbrauch dieser Notation

1. Man schreibt $f = \mathcal{O}(g)$, anstatt $f \in \mathcal{O}(g)$. Dies ist **keine** Gleichheit (wie kann eine Funktion das gleiche wie eine Funktionsmenge sein?).
2. Man schreibt $f(n) = \mathcal{O}(g(n))$, anstatt $f \in \mathcal{O}(g)$, with $f : \mathbb{N} \to \mathbb{R}^+, n \mapsto f(n)$, and $g : \mathbb{N} \to \mathbb{R}^+, n \mapsto g(n)$.
3. Man schreibt $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$, anstatt $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$.

Asymptotische Notation

Man kann einen Ausdruck mit asymptotischer Notation als **Menge** ansehen:

$$n^2 \cdot \mathcal{O}(n) + \mathcal{O}(\log n)$$

repräsentiert

$$\Big\{f : \mathbb{N} \to \mathbb{R}^+ \mid f(n) = n^2 \cdot g(n) + h(n) \\ \text{mit } g(n) \in \mathcal{O}(n) \text{ und } h(n) \in \mathcal{O}(\log n)\Big\}$$

Asymptotische Notation

Lemma 2

Seien f, g *Funktionen mit* $\exists n_0 > 0 \, \forall n \geq n_0 : f(n) > 0$ *(das gleiche für* g*). Dann*

- $c \cdot f(n) = \mathcal{O}(f(n))$ *für eine beliebige Konstante* c
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

Alle obigen Relationen gelten auch für Ω *und* Θ*.*

Asymptotische Notation

Lemma 2

Seien f, g Funktionen mit $\exists n_0 > 0\, \forall n \geq n_0 : f(n) > 0$ (das gleiche für g). Dann

- $c \cdot f(n) = \mathcal{O}(f(n))$ *für eine beliebige Konstante* c
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

Alle obigen Relationen gelten auch für Ω und Θ.

Asymptotische Notation

Lemma 2

Seien f, g Funktionen mit $\exists n_0 > 0 \, \forall n \geq n_0 : f(n) > 0$ (das gleiche für g). Dann

- $c \cdot f(n) = \mathcal{O}(f(n))$ *für eine beliebige Konstante* c
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

Alle obigen Relationen gelten auch für Ω und Θ.

Asymptotische Notation

Lemma 2

Seien f, g *Funktionen mit* $\exists n_0 > 0\, \forall n \geq n_0 : f(n) > 0$ *(das gleiche für* g*). Dann*

- $c \cdot f(n) = \mathcal{O}(f(n))$ *für eine beliebige Konstante* c
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

Alle obigen Relationen gelten auch für Ω *und* Θ*.*

Asymptotische Notation

Lemma 2

Seien f, g *Funktionen mit* $\exists n_0 > 0\, \forall n \geq n_0 : f(n) > 0$ *(das gleiche für* g*). Dann*

- $c \cdot f(n) = \mathcal{O}(f(n))$ *für eine beliebige Konstante* c
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

Alle obigen Relationen gelten auch für Ω *und* Θ*.*

Rechenregel für $\mathcal{O}$-Notation

Zu zeigen:

$$T_1(n) + T_2(n) = \mathcal{O}\big(\max(f(n), g(n))\big)$$

für $T_1(n) \in \mathcal{O}(f(n))$ und $T_2(n) \in \mathcal{O}(g(n))$.

- da $T_1(n) = \mathcal{O}(f(n))$, gibt es $c_1 > 0$ und $n_1 \in \mathbb{N}$ mit $T_1(n) \le c_1 f(n)$ für $n \ge n_1$
- da $T_2(n) = \mathcal{O}(g(n))$, gibt es $c_2 > 0$ und $n_2 \in \mathbb{N}$ mit $T_2(n) \le c_2 g(n)$ für $n \ge n_2$
- Setze $n_0 := \max(n_1, n_2)$, dann ist für $n \ge n_0$

$$\begin{aligned} T_1(n) + T_2(n) &\le c_1 f(n) + c_2 g(n) \\ &\le (c_1 + c_2)\max(f(n), g(n)) \quad \checkmark \end{aligned}$$

Rechenregel für $\mathcal{O}$-Notation

Zu zeigen:

$$T_1(n) + T_2(n) = \mathcal{O}\big(\max(f(n), g(n))\big)$$

für $T_1(n) \in \mathcal{O}(f(n))$ und $T_2(n) \in \mathcal{O}(g(n))$.

- da $T_1(n) = \mathcal{O}(f(n))$, gibt es $c_1 > 0$ und $n_1 \in \mathbb{N}$ mit $T_1(n) \leq c_1 f(n)$ für $n \geq n_1$
- da $T_2(n) = \mathcal{O}(g(n))$, gibt es $c_2 > 0$ und $n_2 \in \mathbb{N}$ mit $T_2(n) \leq c_2 g(n)$ für $n \geq n_2$
- Setze $n_0 := \max(n_1, n_2)$, dann ist für $n \geq n_0$

$$\begin{aligned} T_1(n) + T_2(n) &\leq c_1 f(n) + c_2 g(n) \\ &\leq (c_1 + c_2) \max(f(n), g(n)) \quad \checkmark \end{aligned}$$

Laufzeiten

Funktion	**Eingabelänge n**							
$f(n)$	**10**	10^2	10^3	10^4	10^5	10^6	10^7	10^8
$\log n$	33ns	66ns	0.1µs	0.1µs	0.2µs	0.2µs	0.2µs	0.3µs
$\sqrt{n}$	32ns	0.1µs	0.3µs	1µs	3.1µs	10µs	31µs	0.1ms
n	100ns	1µs	10µs	0.1ms	1ms	10ms	0.1s	1s
$n \log n$	0.3µs	6.6µs	0.1ms	1.3ms	16ms	0.2s	2.3s	27s
$n^{3/2}$	0.3µs	10µs	0.3ms	10ms	0.3s	10s	5.2min	2.7h
n^2	1µs	0.1ms	10ms	1s	1.7min	2.8h	11d	3.2y
n^3	10µs	10ms	10s	2.8h	115d	317y	$3.2 \cdot 10^5$y	
1.1^n	26ns	0.1ms	$7.8 \cdot 10^{25}$y					
2^n	10µs	$4 \cdot 10^{14}$y						
$n!$	36ms	$3 \cdot 10^{142}$y						

1 Operation = 10ns; 100MHz

Alter des Universums: ca. $13.8 \cdot 10^9$y

Typische Laufzeitklassen

$\mathcal{O}$-Notation erlaubt Klassifizierung der Effizienz von Algorithmen

$\Theta(1)$: konstante Laufzeit

- unabhängig von Problemgröße
- *Beispiel:* Löschen von erstem Element in verketteter Liste

$\Theta(\log n)$: logarithmische Laufzeit

- Laufzeit wächst langsamer als Problemgröße
- typisch für Divide & Conquer Algorithmen
- *Beispiel:* Suchen in sortierter Liste

$\Theta(n)$: lineare Laufzeit

- Laufzeit wächst vergleichbar zur Problemgröße
- jedes Eingabe-Element erfordert $\mathcal{O}(1)$ Arbeit
- *Beispiele:* Suchen in unsortierter Liste, Löschen von Element im Array

Typische Laufzeitklassen

> Sofern sich die angegebenen Laufzeiten auf ein Problem beziehen (und nicht auf einen konkreten Algorithmus zu Lösung eines Problems) handelt es sich um einen typischen Lösungsansatz für das jeweilige Problem. Zum Beispiel hat das Standardverfahren für die Matrixmultiplikation eine Laufzeit von $\Theta(n^3)$. Es gibt aber bessere Verfahren.

$\Theta(n \log n)$: "loglinear" Laufzeit

- Laufzeit wächst schneller als Problemgröße
- typisch für Divide & Conquer
- *Beispiele:* Quicksort, FFT

$\Theta(n^2)$: quadratische Laufzeit

- typisch für Algorithmen, die Element paarweise kombinieren
- *Beispiele:* Insertion Sort, Matrix-Vektor Multiplikation

$\Theta(n^3)$: kubische Laufzeit

- *Beispiel:* Matrix-Matrix Multiplikation

> Ein wichtiges offenes Problem ist es ob es z.B. für das TSP-Problem einen Algorithmus mit polynomieller Laufzeit gibt.

$\Theta(2^n)$: exponentielle Laufzeit

- auch als "unlösbar" bezeichnet (intractable)
- *Beispiel:* Traveling Salesman (kürzeste Route, so dass alle Städte exakt einmal besucht)

Asymptotische Notation

Hinweise

- Man sollte asymptotische Notation nicht in Induktionsbeweisen verwenden.
- Für beliebige Konstanten a, b gilt $\log_a n = \Theta(\log_b n)$. Deshalb ignorieren wir die Basis des Algorithmus in asymptotischer Notation.
- Für diese Vorlesung: $\log n = \log_2 n$, d.h., wir nehmen 2 als Standardbasis für den Logarithmus.

Asymptotische Notation

Hinweise

- Man sollte asymptotische Notation nicht in Induktionsbeweisen verwenden.
- Für beliebige Konstanten a, b gilt $\log_a n = \Theta(\log_b n)$. Deshalb ignorieren wir die Basis des Algorithmus in asymptotischer Notation.
- Für diese Vorlesung: $\log n = \log_2 n$, d.h., wir nehmen 2 als Standardbasis für den Logarithmus.

Asymptotische Notation

Hinweise

- Man sollte asymptotische Notation nicht in Induktionsbeweisen verwenden.
- Für beliebige Konstanten a, b gilt $\log_a n = \Theta(\log_b n)$. Deshalb ignorieren wir die Basis des Algorithmus in asymptotischer Notation.
- Für diese Vorlesung: $\log n = \log_2 n$, d.h., wir nehmen 2 als Standardbasis für den Logarithmus.

Asymptotische Notation

Eine asymptotische Klassifizierung von Laufzeiten ist eine gute Möglichkeit um Effizienz von Algorithmen zu vergleichen:

- Falls die Laufzeitanalyse genau genug ist und die Laufzeit auch in der Praxis auftaucht (d.h. keine reine worst-case Schranke), dann ist ein Algorithmus mit besserer asymptotischer Laufzeit einem schwächeren für genügend großes n überlegen.
- **Aber**:
  - Algorithmus A: Laufzeit $f(n) = 1000 \log n = O(\log n)$.
  - Algorithmus B: Laufzeit $g(n) = \log^2 n$.

  Es gilt $f = o(g)$. Aber solange $\log n \le 1000$ ist Algorithmus B effizienter.

Asymptotische Notation

Eine asymptotische Klassifizierung von Laufzeiten ist eine gute Möglichkeit um Effizienz von Algorithmen zu vergleichen:

- Falls die Laufzeitanalyse genau genug ist und die Laufzeit auch in der Praxis auftaucht (d.h. keine reine worst-case Schranke), dann ist ein Algorithmus mit besserer asymptotischer Laufzeit einem schwächeren für genügend großes n überlegen.
- **Aber**:
  - Algorithmus A: Laufzeit $f(n) = 1000 \log n = \mathcal{O}(\log n)$.
  - Algorithmus B: Laufzeit $g(n) = \log^2 n$.

  Es gilt $f = o(g)$. Aber solange $\log n \le 1000$ ist Algorithmus B effizienter.

Asymptotische Notation

Eine asymptotische Klassifizierung von Laufzeiten ist eine gute Möglichkeit um Effizienz von Algorithmen zu vergleichen:

- Falls die Laufzeitanalyse genau genug ist und die Laufzeit auch in der Praxis auftaucht (d.h. keine reine worst-case Schranke), dann ist ein Algorithmus mit besserer asymptotischer Laufzeit einem schwächeren für genügend großes n überlegen.
- **Aber**:
  - Algorithmus A: Laufzeit $f(n) = 1000 \log n = \mathcal{O}(\log n)$.
  - Algorithmus B: Laufzeit $g(n) = \log^2 n$.

Es gilt $f = o(g)$. Aber solange $\log n \leq 1000$ ist Algorithmus B effizienter.

Asymptotische Notation

Eine asymptotische Klassifizierung von Laufzeiten ist eine gute Möglichkeit um Effizienz von Algorithmen zu vergleichen:

- Falls die Laufzeitanalyse genau genug ist und die Laufzeit auch in der Praxis auftaucht (d.h. keine reine worst-case Schranke), dann ist ein Algorithmus mit besserer asymptotischer Laufzeit einem schwächeren für genügend großes n überlegen.
- **Aber**:
  - Algorithmus A: Laufzeit $f(n) = 1000 \log n = \mathcal{O}(\log n)$.
  - Algorithmus B: Laufzeit $g(n) = \log^2 n$.

Es gilt $f = o(g)$. Aber solange $\log n \leq 1000$ ist Algorithmus B effizienter.

Asymptotische Notation

Eine asymptotische Klassifizierung von Laufzeiten ist eine gute Möglichkeit um Effizienz von Algorithmen zu vergleichen:

- Falls die Laufzeitanalyse genau genug ist und die Laufzeit auch in der Praxis auftaucht (d.h. keine reine worst-case Schranke), dann ist ein Algorithmus mit besserer asymptotischer Laufzeit einem schwächeren für genügend großes n überlegen.
- **Aber**:
  - Algorithmus A: Laufzeit $f(n) = 1000 \log n = \mathcal{O}(\log n)$.
  - Algorithmus B: Laufzeit $g(n) = \log^2 n$.

  Es gilt $f = o(g)$. Aber solange $\log n \leq 1000$ ist Algorithmus B effizienter.

Komplexität der elementaren Bausteine

Elementarer Verarbeitungsschritt:

- $\mathcal{O}(1)$

Sequenz:

- Addition in $\mathcal{O}$-Notation

Bedingter Verarbeitungsschritt:

- Maximum von Komplexität von if und else Block, sowie
- $\mathcal{O}(1)$ für Auswertung der Bedingung

Wiederholung:

- Anzahl Wiederholungen multipliziert mit Komplexität Schleifenkörper, sowie
- Anzahl Wiederholungen + 1 multipliziert mit $\mathcal{O}(1)$ für Auswertung der Schleifenbedingung

Komplexität Datenstrukturenoperationen

Feld via Array:

- elementAt $\mathcal{O}(1)$, insert $\mathcal{O}(n)$, erase $\mathcal{O}(n)$

Feld via LinkedList:

- elementAt $\mathcal{O}(n)$, insert $\mathcal{O}(n)$, erase $\mathcal{O}(n)$

Stack via Array:

- push, pop, top alle $\mathcal{O}(1)$

Stack via LinkedList:

- push, pop, top alle $\mathcal{O}(1)$

Queue via LinkedList:

- enqueue, dequeue beide $\mathcal{O}(1)$

Sortieren durch Einfügen

Gegeben: eine Folge von ganzen Zahlen.

Gesucht: die zugehörige aufsteigend sortierte Folge.

Idee:

- speichere die Folge in einem Feld ab;
- lege ein weiteres Feld an;
- füge der Reihe nach jedes Element des ersten Felds an der richtigen Stelle in das zweite Feld ein!

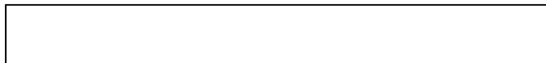
$\Longrightarrow$ Sortieren durch Einfügen (InsertionSort)

Beispiel

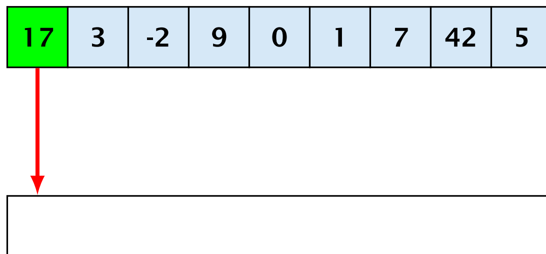
17	3	-2	9	0	1	7	42	5

Beispiel

17	3	-2	9	0	1	7	42	5



Beispiel

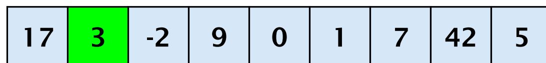


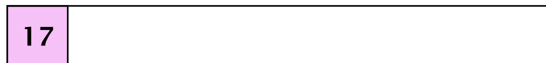
Beispiel

17	3	-2	9	0	1	7	42	5

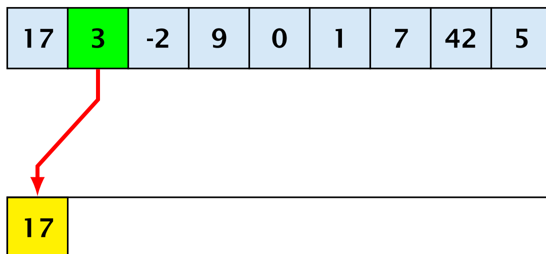
17	

Beispiel

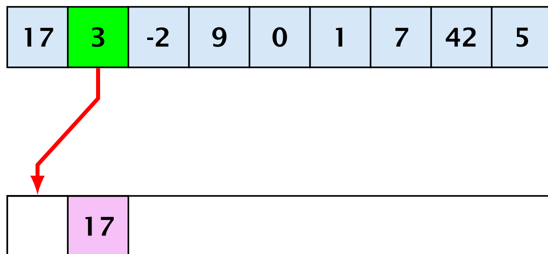




Beispiel



Beispiel

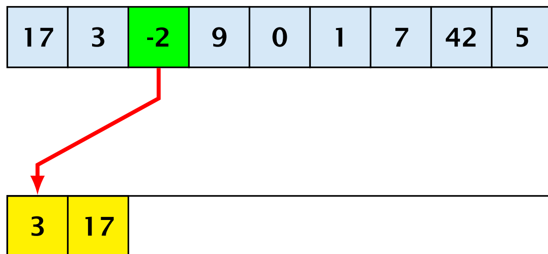


Beispiel

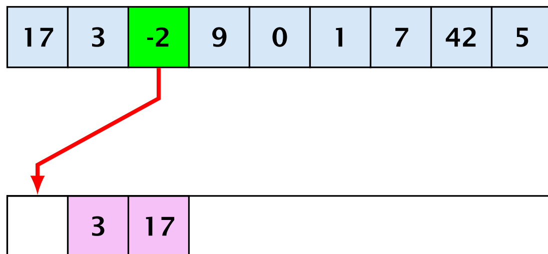
17	3	-2	9	0	1	7	42	5

3	17							

Beispiel



Beispiel

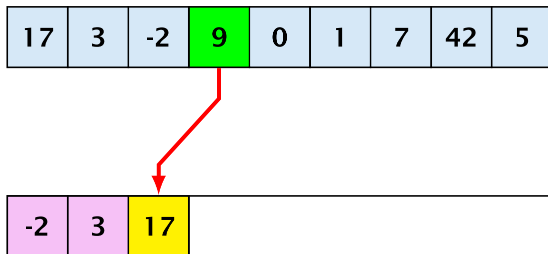


Beispiel

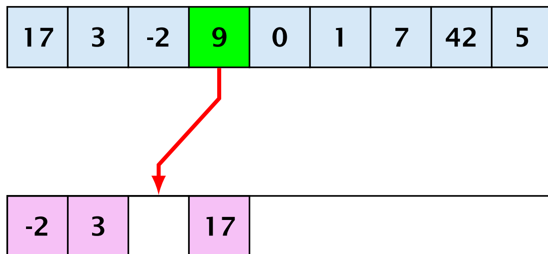
17	3	-2	9	0	1	7	42	5

-2	3	17						

Beispiel



Beispiel

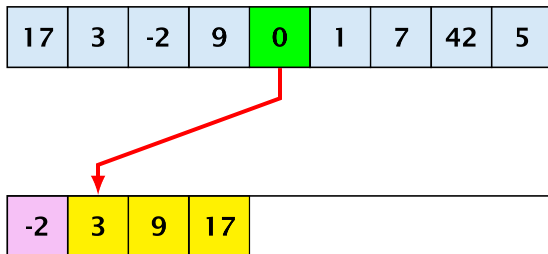


Beispiel

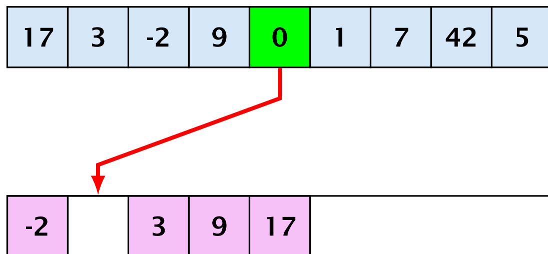
17	3	-2	9	0	1	7	42	5

-2	3	9	17					

Beispiel



Beispiel

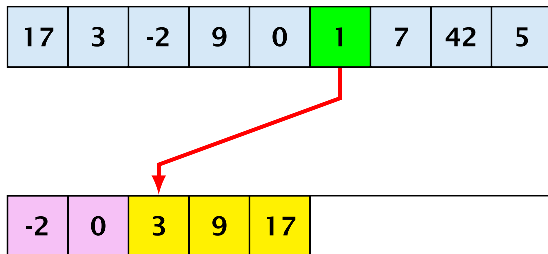


Beispiel

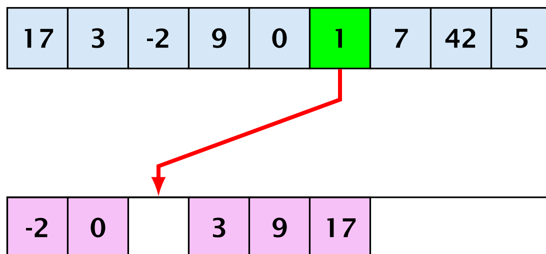
17	3	-2	9	0	1	7	42	5

-2	0	3	9	17				

Beispiel



Beispiel

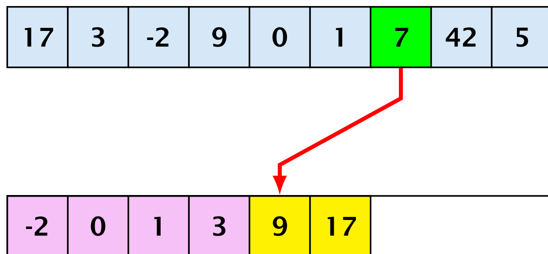


Beispiel

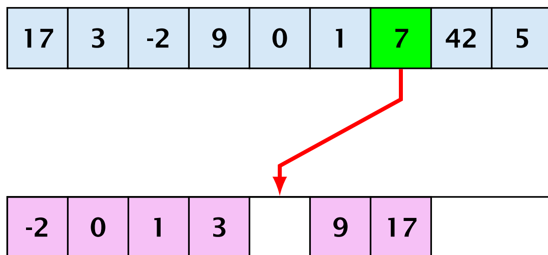
17	3	-2	9	0	1	7	42	5

-2	0	1	3	9	17			

Beispiel



Beispiel

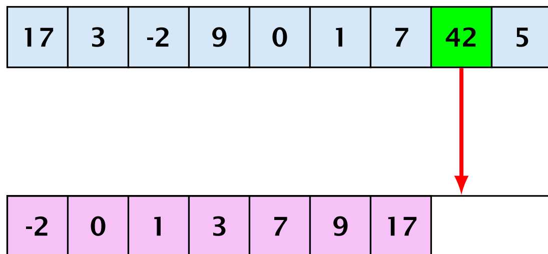


Beispiel

17	3	-2	9	0	1	7	42	5

-2	0	1	3	7	9	17		

Beispiel

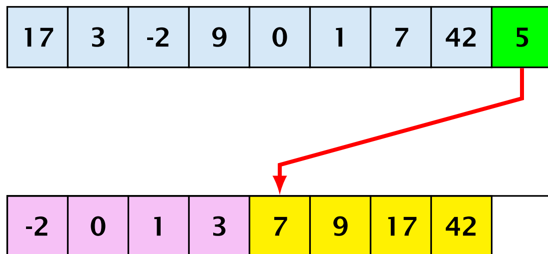


Beispiel

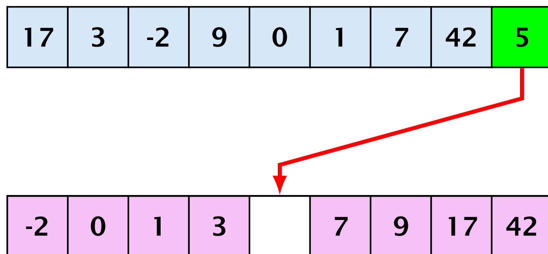
17	3	-2	9	0	1	7	42	5

-2	0	1	3	7	9	17	42	

Beispiel



Beispiel



Beispiel

17	3	-2	9	0	1	7	42	5

-2	0	1	3	5	7	9	17	42

Beispiel

Wir brauchen kein zweites Array:

17	3	-2	9	0	1	7	42	5

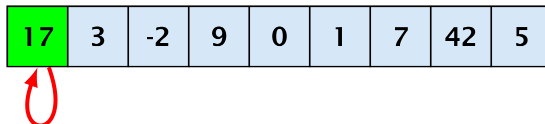
Beispiel

Wir brauchen kein zweites Array:

17	3	-2	9	0	1	7	42	5

Beispiel

Wir brauchen kein zweites Array:



Beispiel

Wir brauchen kein zweites Array:

17	3	-2	9	0	1	7	42	5

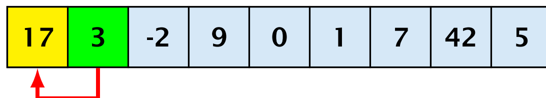
Beispiel

Wir brauchen kein zweites Array:

17	3	-2	9	0	1	7	42	5

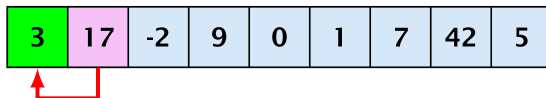
Beispiel

Wir brauchen kein zweites Array:



Beispiel

Wir brauchen kein zweites Array:



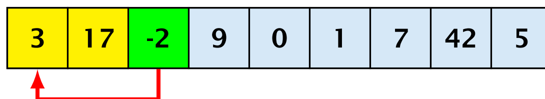
Beispiel

Wir brauchen kein zweites Array:

3	17	-2	9	0	1	7	42	5

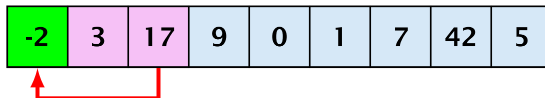
Beispiel

Wir brauchen kein zweites Array:



Beispiel

Wir brauchen kein zweites Array:



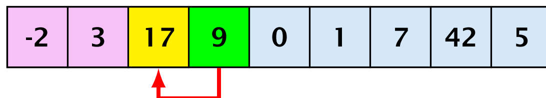
Beispiel

Wir brauchen kein zweites Array:

-2	3	17	9	0	1	7	42	5

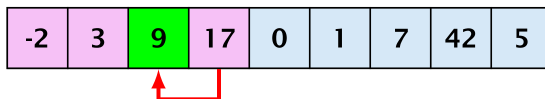
Beispiel

Wir brauchen kein zweites Array:



Beispiel

Wir brauchen kein zweites Array:



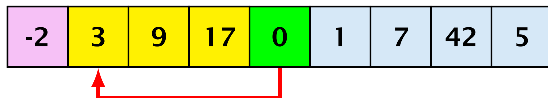
Beispiel

Wir brauchen kein zweites Array:

-2	3	9	17	0	1	7	42	5

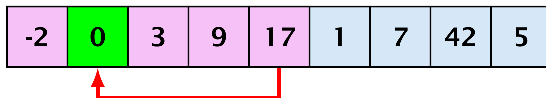
Beispiel

Wir brauchen kein zweites Array:



Beispiel

Wir brauchen kein zweites Array:



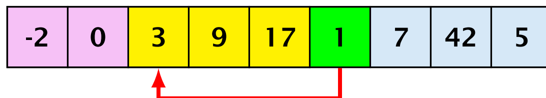
Beispiel

Wir brauchen kein zweites Array:

-2	0	3	9	17	1	7	42	5

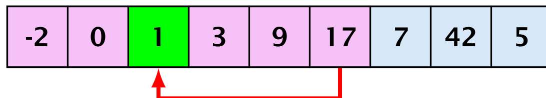
Beispiel

Wir brauchen kein zweites Array:



Beispiel

Wir brauchen kein zweites Array:



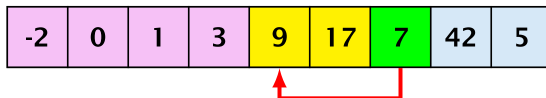
Beispiel

Wir brauchen kein zweites Array:

-2	0	1	3	9	17	7	42	5

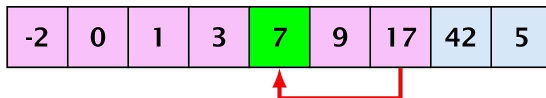
Beispiel

Wir brauchen kein zweites Array:



Beispiel

Wir brauchen kein zweites Array:



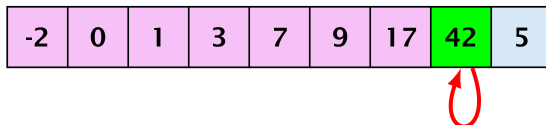
Beispiel

Wir brauchen kein zweites Array:

-2	0	1	3	7	9	17	42	5

Beispiel

Wir brauchen kein zweites Array:



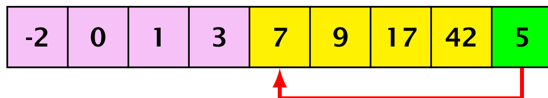
Beispiel

Wir brauchen kein zweites Array:

-2	0	1	3	7	9	17	42	5

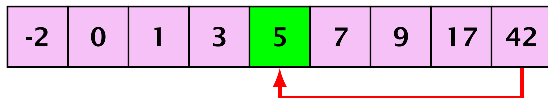
Beispiel

Wir brauchen kein zweites Array:



Beispiel

Wir brauchen kein zweites Array:



Beispiel

Wir brauchen kein zweites Array:

-2	0	1	3	5	7	9	17	42

Algorithmus: Insertion Sort

```
1  Input: Array A mit n Elementen
2  Output: Array A aufsteigend sortiert
3
4  InsertionSort(A)
5      j = 0;
6      while (j < n)
7          key = A[j];
8          i = j-1;
9          while (i >= 0 && A[i] > key)
10             A[i+1] = A[i];
11             i--;
12         A[i+1] = key;
13         j++;
```

InsertionSort

Laufzeit InsertionSort

Kostenübersicht		
Zeile	**Kosten**	**Anzahl**
5	$\mathcal{O}(1)$	1
6	$\mathcal{O}(1)$	$n+1$
7	$\mathcal{O}(1)$	$n-1$
8	$\mathcal{O}(1)$	t_j
9	$\mathcal{O}(1)$	t_j-1
10	$\mathcal{O}(1)$	t_j-1
11	$\mathcal{O}(1)$	n
12	$\mathcal{O}(1)$	n

```
1  Input: Array A mit Laenge n
2  Output: sortiertes Array
3
4  InsertionSort(A)
5    j = 0;
6    while (j < n)
7      key = A[j];
8      i = j-1;
9      while (i>=0 && A[i]>key)
10       A[i+1] = A[i];
11       i--;
12     A[i+1] = key;
13     j++;
```

t_j bezeichnet Anzahl der Abfragen der while-Bedingung in Zeile 8 für Durchlauf j

Laufzeit InsertionSort

$$
\begin{aligned}
&\mathcal{O}(1) + (n+1)\mathcal{O}(1) + (n-1)\mathcal{O}(1) + \mathcal{O}(1)\sum_{j=0}^{n-1} t_j + \\
&\qquad \mathcal{O}(1)\sum_{j=0}^{n-1}(t_j - 1) + \mathcal{O}(1)\sum_{j=0}^{n-1}(t_j - 1) + n\mathcal{O}(1) + n\mathcal{O}(1) \\
&= \mathcal{O}(1)n + \mathcal{O}(1)\sum_{j=0}^{n-1} t_j \\
&= \mathcal{O}\left(n + \sum_{j=0}^{n-1} t_j\right)
\end{aligned}
$$

Die Laufzeit hängt stark vom Input ab.

Laufzeit InsertionSort

Best-case:
Wenn das Array sortiert wird ist, ist $t_j = 1$.
$\Longrightarrow$ Laufzeit: $\mathcal{O}(n)$.

Worst-case:
Wenn das Array absteigend sortiert ist, ist $t_j = j + 1$.
$\Longrightarrow$ Laufzeit: $\mathcal{O}(n^2)$.

Beobachtung:
Wenn ein Element höchstens h Positionen von seiner Zielposition entfernt ist, dann ist $t_j \leq h + 1$.
$\Longrightarrow$ Laufzeit: $\mathcal{O}(hn)$.

5 Effizienz von Algorithmen

Gegeben: Array `A` ganzer Zahlen; Element `x`

Gesucht: Wo kommt `x` in `A` vor?

Naives Vorgehen:

- Vergleiche `x` der Reihe nach mit `A[0]`, `A[1]`, usw.
- Finden wir `i` mit `A[i] == x`, geben wir `i` aus.
- Andernfalls geben wir `-1` aus: „Element nicht gefunden"!

Naives Suchen

```
1  Input:  Array A mit Laenge n; Element x
2  Output: i mit A[i] == x falls existent
3          sonst -1
4
5  find(A,x)
6      i = 0;
7      while (i < n && A[i] != x)
8          i++;
9      if (i == n)
10         return -1;
11     else
12         return i;
```

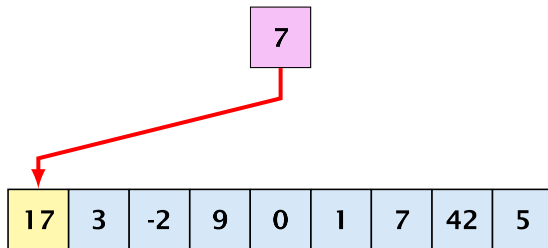
Naives Suchen

Beispiel

7

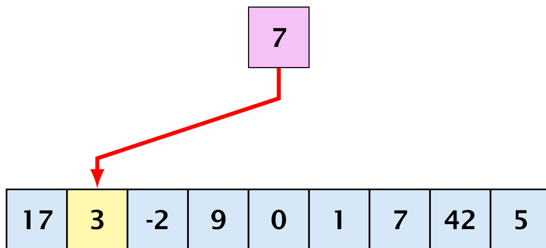
17	3	-2	9	0	1	7	42	5

Beispiel

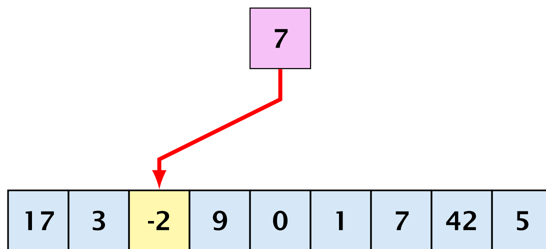


no

Beispiel

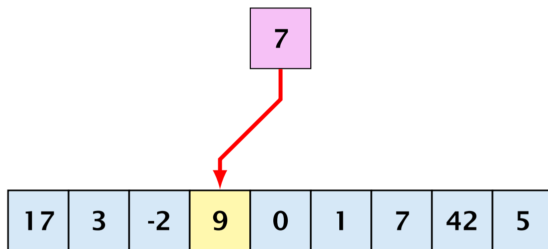


Beispiel



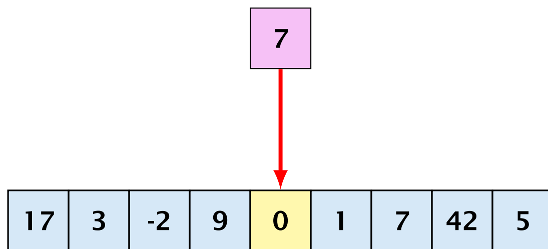
no

Beispiel

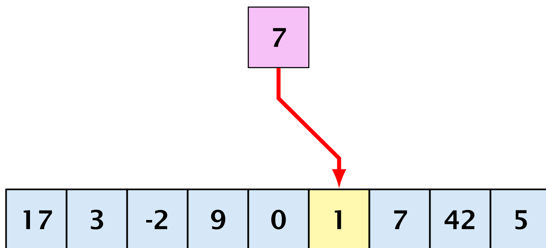


no

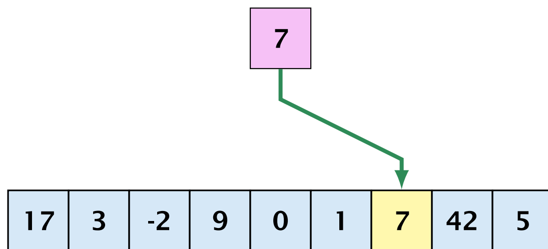
Beispiel



Beispiel



Beispiel



yes

Laufzeit Naives Suchen

Best-case:
Wenn x an Position 0.
$\Longrightarrow$ Laufzeit: $\mathcal{O}(1)$.

Worst-case:
Wenn x nicht vorkommt.
$\Longrightarrow$ Laufzeit: $\mathcal{O}(n)$.

... geht das besser?

Binäre Suche

Annahme: Input ist sortiert.

Idee:

- Vergleiche x mit dem Wert, der in der Mitte steht.
- Liegt Gleichheit vor, sind wir fertig.
- Ist x kleiner, brauchen wir nur noch links weitersuchen.
- Ist x größer, brauchen wir nur noch rechts weiter suchen.

$\Longrightarrow$ binäre Suche

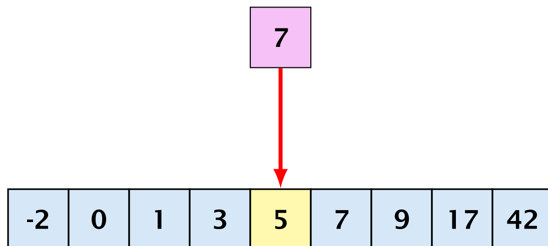
Beispiel

7

-2	0	1	3	5	7	9	17	42

- wir benötigen nur drei Vergleiche

Beispiel



no

- wir benötigen nur drei Vergleiche

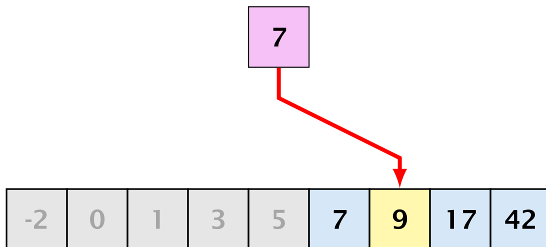
Beispiel

7

-2	0	1	3	5	7	9	17	42

- wir benötigen nur drei Vergleiche

Beispiel



no

- wir benötigen nur drei Vergleiche

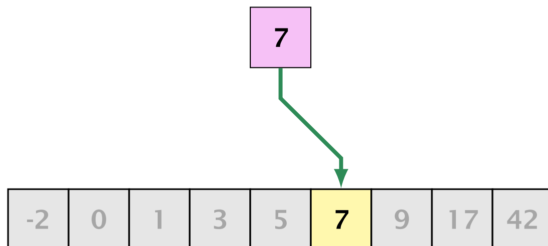
Beispiel

7

-2	0	1	3	5	7	9	17	42

- wir benötigen nur drei Vergleiche

Beispiel



- wir benötigen nur drei Vergleiche

Implementierung

```
1  Input:  sortiertes Array A; Element x;
2          linker Index nl; rechter Index nr; nl<=nr
3  Output: Index i; nl ≤ i ≤ nr mit A[i]==x falls existent
4          sonst -1
5  find(A, x, nl, nr) // Inputlaenge ist n=nr-nl+1
6      t = (nl + nr) / 2;
7      if (A[t] == x)
8          return t;
9      if (nl == nr)
10         return -1;
11     if (x > A[t])
12         return find(A, x, t+1, nr);
13     if (nl < t)
14         return find(A, x, nl, t-1);
15     return -1;
```

Laufzeit Binäre Suche

Laufzeit:

$$T(n) = \begin{cases} \mathcal{O}(1) & \texttt{A[t] == x} \\ \mathcal{O}(1) & \texttt{nl == nr} \\ \mathcal{O}(1) + T(nr - t) & \texttt{x > A[t]} \\ \mathcal{O}(1) + T(t - nl) & \texttt{nl < t} \end{cases}$$

oder

$$T(n) \leq \begin{cases} \mathcal{O}(1) & n = 1 \\ \mathcal{O}(1) + T(\lfloor n/2 \rfloor) & \text{sonst} \end{cases}$$

Laufzeit Binäre Suche

Lösen der Rekursionsgleichung:

$$T(n) \leq \begin{cases} \mathcal{O}(1) & n = 1 \\ \mathcal{O}(1) + T(\lfloor n/2 \rfloor) & \text{sonst} \end{cases}$$

Üblicherweise nur für $n = 2^k$; z.B. durch vollständige Induktion.

Rekurrenzen

Wie finden wir einen geschlossenen Ausdruck für die Laufzeit?

Dafür müssen wir die Rekursionsgleichung lösen.

Rekurrenzen

Wie finden wir einen geschlossenen Ausdruck für die Laufzeit?

Dafür müssen wir die Rekursionsgleichung lösen.

Methoden

1. **Raten+Induktion**
   Man rät die richtige Lösung und beweist die Korrektheit mittels vollständiger Induktion. Man benötigt Erfahrung um richtig zu raten...
2. **Mastertheorem**
   Für die meisten Rekurrenzen gibt es ein allgemeines Theorem, dass die asymptotisch korrekte Laufzeit für die jeweilige Rekurrenz angibt.

Raten+Induktion

Zuerst müssen wir die $\mathcal{O}$-Notation entfernen:

$$T(n) \leq \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + c_1 n & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Raten+Induktion

Zuerst müssen wir die $\mathcal{O}$-Notation entfernen:

$$T(n) \le \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + c_1 n & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

für $n = 2^k$:

Für diesen Fall können wir stattdessen die folgende Rekursionsgleichung betrachten:

$$T(n) \le \begin{cases} T(\frac{n}{2}) + c_1 n & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Raten+Induktion

Zuerst müssen wir die $\mathcal{O}$-Notation entfernen:

$$T(n) \leq \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + c_1 n & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

für $n = 2^k$:
Für diesen Fall können wir stattdessen die folgende Rekursionsgleichung betrachten:

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 n & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Man rät die richtige Lösung und beweist, dass durch Einsetzen, dass diese Lösung korrekt ist.

Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \leq a \log n + b$.

Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \leq a \log n + b$.

Beweis. (durch Induktion)

Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \leq a \log n + b$.

Beweis. (durch Induktion)

- **Anfang** $(n = 1)$:

Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \leq a \log n + b$.

Beweis. (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.

Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \leq a \log n + b$.

Beweis. (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.
- **Induktionsschritt** $1, \dots, n-1 \to n$:

Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \leq a \log n + b$.

Beweis. (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für n:

Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \leq a \log n + b$.

Beweis. (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für n:

$$T(n) \leq T\left(\frac{n}{2}\right) + c_1$$

Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \leq a \log n + b$.

Beweis. (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für n:

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2}\right) + c_1 \\ &\leq \left(a \log \frac{n}{2} + b\right) + c_1 \end{aligned}$$

Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \leq a \log n + b$.

Beweis. (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für n:

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2}\right) + c_1 \\ &\leq \left(a \log \frac{n}{2} + b\right) + c_1 \\ &= a(\log n - 1) + b + c_1 \end{aligned}$$

Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \leq a \log n + b$.

Beweis. (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für n:

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2}\right) + c_1 \\ &\leq \left(a \log \frac{n}{2} + b\right) + c_1 \\ &= a(\log n - 1) + b + c_1 \\ &= a \log n + (c_1 - a) + b \end{aligned}$$

Raten+Induktion

$$T(n) \le \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \le a \log n + b$.

Beweis. (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \ge c_2$.
- **Induktionsschritt** $1, \dots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \dots, n-1\}$, und $n > 1$. Wir beweisen sie für n:

$$\begin{aligned} T(n) &\le T\left(\frac{n}{2}\right) + c_1 \\ &\le \left(a \log \frac{n}{2} + b\right) + c_1 \\ &= a(\log n - 1) + b + c_1 \\ &= a \log n + (c_1 - a) + b \\ &\le a \log n + b \end{aligned}$$

Raten+Induktion

$$T(n) \leq \begin{cases} T(\frac{n}{2}) + c_1 & n > 1 \\ c_2 & \text{sonst} \end{cases}$$

Ansatz: $T(n) \leq a \log n + b$.

Beweis. (durch Induktion)

- **Anfang** $(n = 1)$: wahr falls $b \geq c_2$.
- **Induktionsschritt** $1, \ldots, n-1 \to n$:

  Angenommen Aussage wahr für $n' \in \{1, \ldots, n-1\}$, und $n > 1$. Wir beweisen sie für n:

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2}\right) + c_1 \\ &\leq \left(a \log \frac{n}{2} + b\right) + c_1 \\ &= a(\log n - 1) + b + c_1 \\ &= a \log n + (c_1 - a) + b \\ &\leq a \log n + b \end{aligned}$$

  Gilt falls $a \geq c_1$.

Mastertheorem

Lemma 3

Seien $a \geq 1, b \geq 1$ *und* $\epsilon > 0$ *Konstanten. Betrachte die Rekurrenz*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \ .$$

1. Fall:

Falls $f(n) = \mathcal{O}(n^{\log_b(a)-\epsilon})$ *gilt* $T(n) = \Theta(n^{\log_b a})$.

2. Fall:

Falls $f(n) = \Theta(n^{\log_b(a)} \log^k n)$ *gilt* $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$, $k \geq 0$.

Mastertheorem

Wir beweisen das Mastertheorem für den Fall $n = b^\ell$, und nehmen an, dass der nichtrekursive Fall für Problemgröße 1 Kosten 1 verursacht.

Der Rekursionsbaum

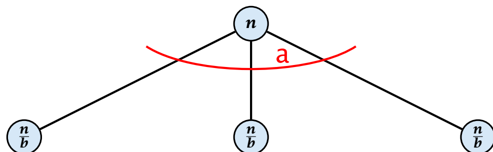
Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:

Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:

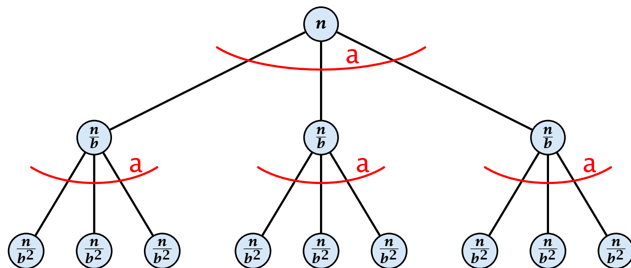
Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



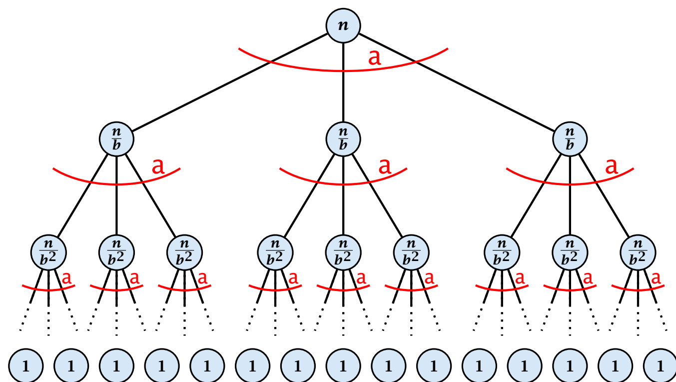
Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



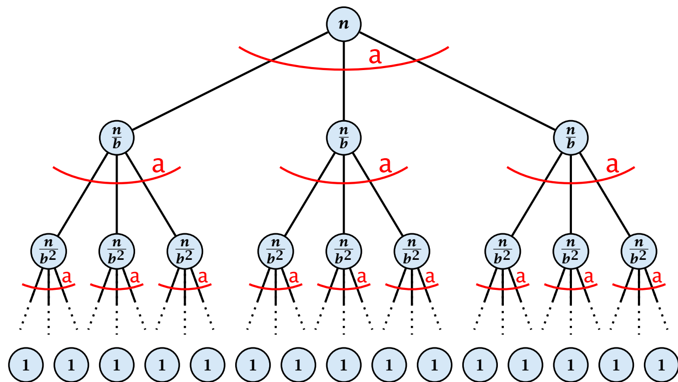
Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



Der Rekursionsbaum

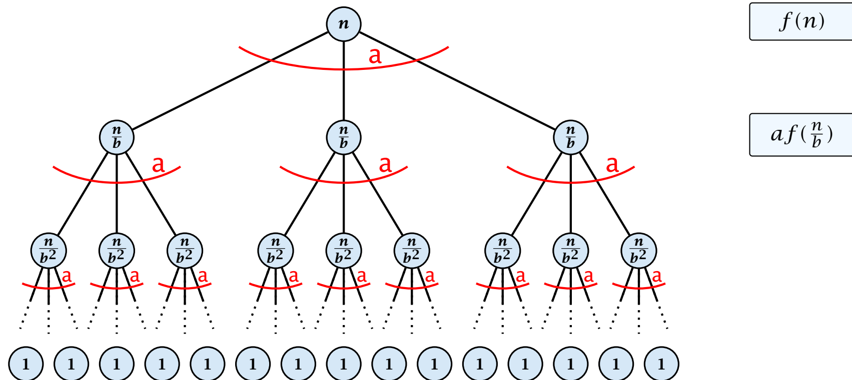
Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



$f(n)$

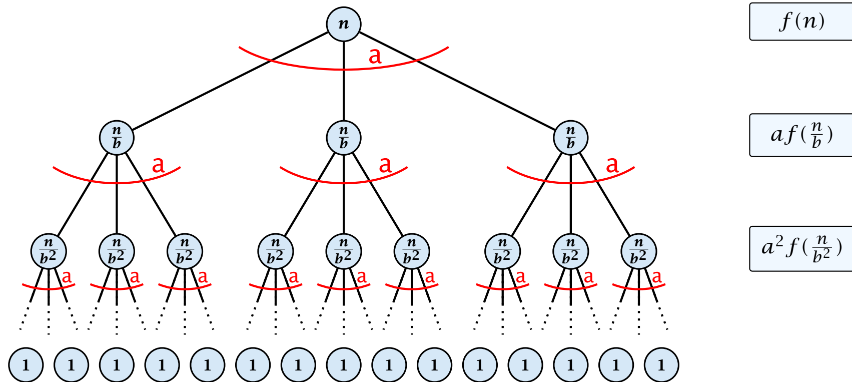
Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



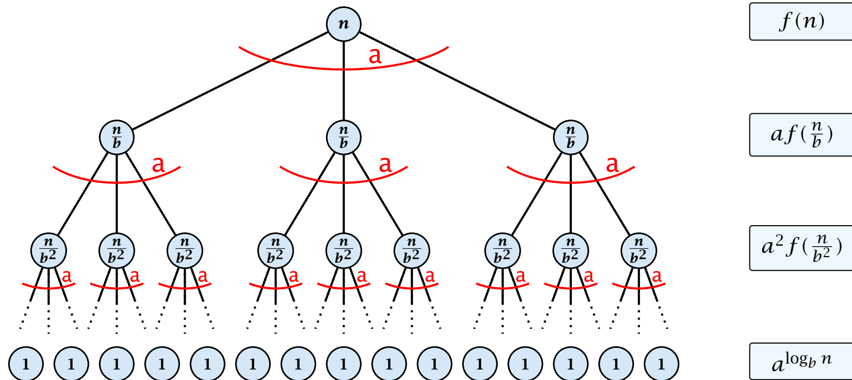
Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



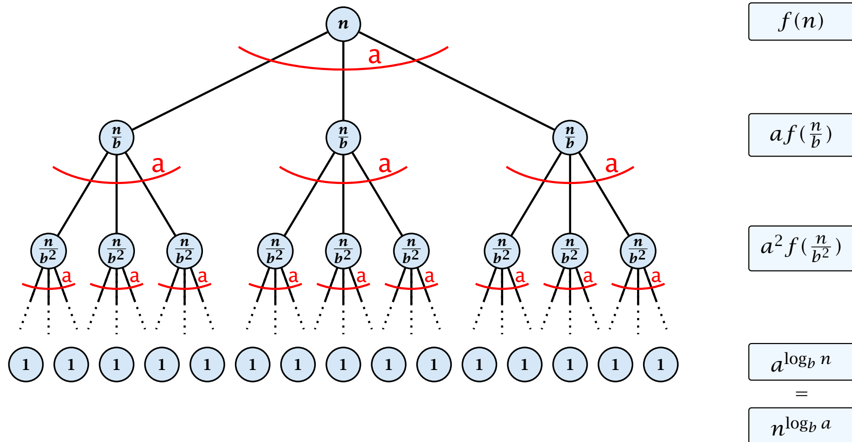
Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



Der Rekursionsbaum

Die Laufzeit eines rekursiven Algorithmus kann durch einen Rekursionsbaum veranschaulicht werden:



Mastertheorem

Das heißt unsere Kosten sind

$$T(n) = n^{\log_b a} + \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \; .$$

Fall 1. Sei $f(n) \leq cn^{\log_b a - \epsilon}$.

Fall 1. Sei $f(n) \leq cn^{\log_b a-\epsilon}$.

$$T(n) - n^{\log_b a}$$

Fall 1. Sei $f(n) \leq cn^{\log_b a - \epsilon}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

Fall 1. Sei $f(n) \leq cn^{\log_b a - \epsilon}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon}
\end{aligned}
$$

Fall 1. Sei $f(n) \leq cn^{\log_b a - \epsilon}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon}
\end{aligned}
$$

$$b^{-i(\log_b a - \epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}$$

Fall 1. Sei $f(n) \leq cn^{\log_b a - \epsilon}$.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \\ &= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} \left(b^{\epsilon}\right)^i \end{aligned}$$

$b^{-i(\log_b a - \epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}$

Fall 1. Sei $f(n) \leq cn^{\log_b a - \epsilon}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \\
&= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} \left(b^{\epsilon}\right)^i
\end{aligned}
$$

$b^{-i(\log_b a - \epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}$

$\sum_{i=0}^{k} q^i = \frac{q^{k+1} - 1}{q - 1}$

Fall 1. Sei $f(n) \leq cn^{\log_b a - \epsilon}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon}$$

$b^{-i(\log_b a - \epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}$

$$= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} (b^{\epsilon})^i$$

$\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}$

$$= cn^{\log_b a - \epsilon}(b^{\epsilon \log_b n} - 1)/(b^{\epsilon} - 1)$$

Fall 1. Sei $f(n) \le cn^{\log_b a-\epsilon}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\le c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon} \\
&= cn^{\log_b a-\epsilon} \sum_{i=0}^{\log_b n-1} \left(b^{\epsilon}\right)^i \\
&= cn^{\log_b a-\epsilon}(b^{\epsilon \log_b n} - 1)/(b^{\epsilon} - 1) \\
&= cn^{\log_b a-\epsilon}(n^{\epsilon} - 1)/(b^{\epsilon} - 1)
\end{aligned}
$$

$b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i}a^{-i}$

$\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}$

Fall 1. Sei $f(n) \leq cn^{\log_b a - \epsilon}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon}$$

$b^{-i(\log_b a - \epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}$

$$= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} (b^\epsilon)^i$$

$\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}$

$$= cn^{\log_b a - \epsilon}(b^{\epsilon \log_b n} - 1)/(b^\epsilon - 1)$$

$$= cn^{\log_b a - \epsilon}(n^\epsilon - 1)/(b^\epsilon - 1)$$

$$= \frac{c}{b^\epsilon - 1} n^{\log_b a}(n^\epsilon - 1)/(n^\epsilon)$$

Fall 1. Sei $f(n) \leq cn^{\log_b a-\epsilon}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon} \\
&= cn^{\log_b a-\epsilon} \sum_{i=0}^{\log_b n-1} \left(b^\epsilon\right)^i \\
&= cn^{\log_b a-\epsilon}(b^{\epsilon \log_b n} - 1)/(b^\epsilon - 1) \\
&= cn^{\log_b a-\epsilon}(n^\epsilon - 1)/(b^\epsilon - 1) \\
&= \frac{c}{b^\epsilon - 1} n^{\log_b a}(n^\epsilon - 1)/(n^\epsilon)
\end{aligned}
$$

$b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}$

$\sum_{i=0}^k q^i = \frac{q^{k+1}-1}{q-1}$

Also,

$$
T(n) \leq \left(\frac{c}{b^\epsilon - 1} + 1\right) n^{\log_b (a)}
$$

Fall 1. Sei $f(n) \leq cn^{\log_b a - \epsilon}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \\
&= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} \left(b^{\epsilon}\right)^i \\
&= cn^{\log_b a - \epsilon}(b^{\epsilon \log_b n} - 1)/(b^{\epsilon} - 1) \\
&= cn^{\log_b a - \epsilon}(n^{\epsilon} - 1)/(b^{\epsilon} - 1) \\
&= \frac{c}{b^{\epsilon} - 1} n^{\log_b a}(n^{\epsilon} - 1)/(n^{\epsilon})
\end{aligned}
$$

$b^{-i(\log_b a - \epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}$

$\sum_{i=0}^{k} q^i = \frac{q^{k+1} - 1}{q - 1}$

Also,

$$
T(n) \leq \left(\frac{c}{b^{\epsilon} - 1} + 1\right) n^{\log_b (a)}
$$

$\Rightarrow T(n) = \mathcal{O}(n^{\log_b a})$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}$.

$$T(n) - n^{\log_b a}$$

Fall 2. Sei $f(n) \le cn^{\log_b a}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

Fall 2. Sei $f(n) \leq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a}
\end{aligned}
$$

Fall 2. Sei $f(n) \leq cn^{\log_b a}$.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\ &= cn^{\log_b a} \sum_{i=0}^{\log_b n-1} 1 \end{aligned}$$

Fall 2. Sei $f(n) \leq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\
&= cn^{\log_b a} \log_b n
\end{aligned}
$$

Fall 2. Sei $f(n) \le cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\le c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\
&= cn^{\log_b a} \log_b n
\end{aligned}
$$

Also,

$$T(n) = \mathcal{O}(n^{\log_b a} \log_b n)$$

Fall 2. Sei $f(n) \leq cn^{\log_b a}$.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\ &= cn^{\log_b a} \sum_{i=0}^{\log_b n-1} 1 \\ &= cn^{\log_b a} \log_b n \end{aligned}$$

Also,

$$T(n) = \mathcal{O}(n^{\log_b a} \log_b n)$$

$\Rightarrow T(n) = \mathcal{O}(n^{\log_b a} \log n)$.

Fall 2. Sei $f(n) \geq cn^{\log_b a}$.

Fall 2. Sei $f(n) \geq cn^{\log_b a}$.

$$T(n) - n^{\log_b a}$$

Fall 2. Sei $f(n) \geq cn^{\log_b a}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

Fall 2. Sei $f(n) \geq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a}
\end{aligned}
$$

Fall 2. Sei $f(n) \geq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1
\end{aligned}
$$

Fall 2. Sei $f(n) \geq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\
&= cn^{\log_b a} \log_b n
\end{aligned}
$$

Fall 2. Sei $f(n) \geq cn^{\log_b a}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a}$$

$$= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1$$

$$= cn^{\log_b a} \log_b n$$

Also,

$$T(n) = \Omega(n^{\log_b a} \log_b n)$$

Fall 2. Sei $f(n) \geq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\geq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n-1} 1 \\
&= cn^{\log_b a} \log_b n
\end{aligned}
$$

Also,

$$
T(n) = \Omega(n^{\log_b a} \log_b n) \qquad \Rightarrow T(n) = \Omega(n^{\log_b a} \log n).
$$

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a}$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k
\end{aligned}
$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k$$

$n = b^\ell \Rightarrow \ell = \log_b n$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k$$

$n = b^\ell \Rightarrow \ell = \log_b n$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k
\end{aligned}
$$

$n = b^\ell \Rightarrow \ell = \log_b n$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde. Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \le cn^{\log_b a}(\log_b(n))^k$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\le c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell - 1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell - 1} (\ell - i)^k \\
&= cn^{\log_b a} \sum_{i=1}^{\ell} i^k
\end{aligned}
$$

$n = b^\ell \Rightarrow \ell = \log_b n$

Beachte, dass $\sum_{i=1}^{\ell} i^k \le \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde.
Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \ge \sum_{i=\ell/2}^{\ell} i^k \ge (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k$$

$n = b^\ell \Rightarrow \ell = \log_b n$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k$$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k$$

$$= cn^{\log_b a} \sum_{i=1}^{\ell} i^k \approx \frac{1}{k}\ell^{k+1}$$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde. Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k \\
&= cn^{\log_b a} \sum_{i=1}^{\ell} i^k \\
&\approx \frac{c}{k} n^{\log_b a} \ell^{k+1}
\end{aligned}
$$

$n = b^\ell \Rightarrow \ell = \log_b n$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde. Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

Fall 2. Sei $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k \\
&= cn^{\log_b a} \sum_{i=1}^{\ell} i^k \\
&\approx \frac{c}{k} n^{\log_b a} \ell^{k+1}
\end{aligned}
$$

$n = b^\ell \Rightarrow \ell = \log_b n$

Beachte, dass $\sum_{i=1}^{\ell} i^k \leq \ell^{k+1}$ trivial ist, und für diese obere Schranke ausreichen würde. Für eine untere Schranke reicht auch $\sum_{i=1}^{\ell} i^k \geq \sum_{i=\ell/2}^{\ell} i^k \geq (\ell/2)^{k+1}$.

$\Rightarrow T(n) = \mathcal{O}(n^{\log_b a} \log^{k+1} n)$.

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

$$
\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
\hline
\end{array}
$$

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

$$
\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
\hline
\end{array}
$$

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

1	1	0	1	1	0	1	0	1	A
1	0	0	0	1	0	0	1	1	B
							1		
								0	

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

1	1	0	1	1	0	1	0	1	A
1	0	0	0	1	0	0	1_1	1	B
								0	

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

$$
\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0_{1} & 1_{1} & 1 & & B \\
\hline
 & & & & & & & 0 & 0 & &
\end{array}
$$

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

$$
\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0_{1} & 1_{1} & 1 & & B \\
\hline
 & & & & & & & 0 & 0 & &
\end{array}
$$

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

1	1	0	1	1	0	1	0	1	A
1	0	0	0	1	0	0	1	1	B
					1	1	1		
						0	0	0	

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

1	1	0	1	1	0	1	0	1	A
1	0	0	0	1	0	0	1	1	B
					1	1	1		
						0	0	0	

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

1	1	0	1	1	0	1	0	1	A
1	0	0	0	1	0	0	1	1	B
				0	1	1	1		
					1	0	0	0	

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

1	1	0	1	1	0	1	0	1	A
1	0	0	0	1	0	0	1	1	B
				0	1	1	1		
					1	0	0	0	

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

1	1	0	1	1	0	1	0	1	A
1	0	0	0	1	0	0	1	1	B
			1	0	1	1	1		
				0	1	0	0	0	

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

1	1	0	1	1	0	1	0	1	A
1	0	0	0	1	0	0	1	1	B
			1	0	1	1	1		
				0	1	0	0	0	

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

1	1	0	1	1	0	1	0	1	A
1	0	0	0	1	0	0	1	1	B
		1	1	0	1	1	1		
			0	0	1	0	0	0	

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

$$
\begin{array}{ccccccccccc}
 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\
 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\
 & & & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\
\hline
 & & & & 0 & 0 & 1 & 0 & 0 & 0 &
\end{array}
$$

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

1	1	0	1	1	0	1	0	1	A
1	0	0	0	1	0	0	1	1	B
	0	1	1	0	1	1	1		
		1	0	0	1	0	0	0	

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

1	1	0	1	1	0	1	0	1	A
1	0	0	0	1	0	0	1	1	B
	0	1	1	0	1	1	1		
		1	0	0	1	0	0	0	

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

1	1	0	1	1	0	1	0	1	A
1	0	0	0	1	0	0	1	1	B
0	0	1	1	0	1	1	1		
	1	1	0	0	1	0	0	0	

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

$$\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
{}_0 & {}_0 & {}_1 & {}_1 & {}_0 & {}_1 & {}_1 & {}_1 & & & \\
\hline
 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & &
\end{array}$$

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

	1	1	0	1	1	0	1	0	1	A
	1	0	0	0	1	0	0	1	1	B
1	0	0	1	1	0	1	1	1		
	0	1	1	0	0	1	0	0	0	

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

	1	1	0	1	1	0	1	0	1	A
	1	0	0	0	1	0	0	1	1	B
1	0	0	1	1	0	1	1	1		
	0	1	1	0	0	1	0	0	0	

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

	1	1	0	1	1	0	1	0	1	A
	1	0	0	0	1	0	0	1	1	B
1	0	0	1	1	0	1	1	1		
1	0	1	1	0	0	1	0	0	0	

Beispiel: Integermultiplikation

Angenommen wir möchten zwei n-bit Integer multiplizieren, aber unsere Register können nur Operationen auf Integern konstanter Länge ausführen.

Dafür müssen wir zunächst zwei Zahlen A and B addieren können:

$$
\begin{array}{cccccccccccl}
 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
 {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & \\
\hline
1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & &
\end{array}
$$

Das heißt wir können zwei n-bit Integer in Zeit $\mathcal{O}(n)$ addieren.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ 1 \\ \hline \end{array}$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$1\ 0\ 0\ 0\ 1 \times 1\ 0\ 1\ \boxed{1}$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{r}
1\;0\;0\;0\;1 \times 1\;0\;1\;\boxed{1} \\
\hline
1\;0\;0\;0\;1
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{r}
1\ 0\ 0\ 0\ 1 \times 1\ 0\ \boxed{1}\ 1 \\
\hline
1\ 0\ 0\ 0\ 1
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{r}
1\ 0\ 0\ 0\ 1 \times 1\ 0\ \boxed{1}\ 1 \\
\hline
1\ 0\ 0\ 0\ 1 \\
0
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

```
1 0 0 0 1 × 1 0 [1] 1
---------------------
          1 0 0 0 1
        1 0 0 0 1 0
```

- Dies nennt man auch die „Schulmethode" für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{r}
1\ 0\ 0\ 0\ 1 \times 1\ \boxed{0}\ 1\ 1 \\
\hline
1\ 0\ 0\ 0\ 1 \\
1\ 0\ 0\ 0\ 1\ 0
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{r}
1\;0\;0\;0\;1 \times 1\;\boxed{0}\;1\;1 \\
\hline
1\;0\;0\;0\;1 \\
1\;0\;0\;0\;1\;0 \\
0\;0
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

1	0	0	0	1	×	1	0	1	1
					1	0	0	0	1
				1	0	0	0	1	0
			0	0	0	0	0	0	0

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{rrrrrrrrrr}
1 & 0 & 0 & 0 & 1 & \times & \boxed{1} & 0 & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

- Dies nennt man auch die „Schulmethode" für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

1	0	0	0	1	×	1	0	1	1
						1	0	0	0	1
					1	0	0	0	1	0
			0	0	0	0	0	0	0
							0	0	0

- Dies nennt man auch die „Schulmethode" für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{rrrrrrrrrrr}
 & & & 1 & 0 & 0 & 0 & 1 & \times & \boxed{1} & 0 & 1 & 1 \\
\hline
 & & & & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{rrrrrrrrr}
 & 1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{rrrrrrrrr}
 & 1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
 & & & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \le n$) multiplizieren.

$$
\begin{array}{ccccccccc}
 & & 1 & 0 & 0 & 0 & 1 & \times & 1\ 0\ 1\ 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0\ 1 \\
 & & & & 1 & 0 & 0 & 0 & 1\ 0 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0\ 0 \\
 & & 1 & 0 & 0 & 0 & 1 & 0 & 0\ 0 \\
\hline
 & & 1 & 0 & 1 & 1 & 1 & 0 & 1\ 1
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \le 2n$ bits.

Laufzeit:

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{r}
10001 \times 1011 \\
\hline
10001 \\
100010 \\
0000000 \\
10001000 \\
\hline
10111011
\end{array}
$$

- Dies nennt man auch die „Schulmethode" für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Laufzeit:

- Zwischenergebnisse berechnen: $\mathcal{O}(nm)$.

Beispiel: Integermultiplikation

Angenommen wir möchten ein n-bit Integer A und ein m-bit Integer B ($m \leq n$) multiplizieren.

$$
\begin{array}{cccccccc}
 & & 1 & 0 & 0 & 0 & 1 & \times\ 1\ 0\ 1\ 1 \\
\hline
 & & & 1 & 0 & 0 & 0 & 1 \\
 & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1
\end{array}
$$

- Dies nennt man auch die „Schulmethode“ für die Integermultiplikation.
- Die Zahlen der Zwischenergebenisse haben höchstens $m + n \leq 2n$ bits.

Laufzeit:

- Zwischenergebnisse berechnen: $\mathcal{O}(nm)$.
- Addieren von m Zahlen der Länge $\leq 2n$: $\mathcal{O}((m+n)m) = \mathcal{O}(nm)$.

Beispiel: Integermultiplikation

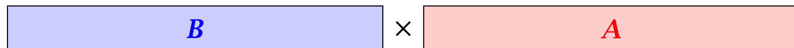
Ein rekursiver Ansatz:

Angenommen die Integer A und B haben Länge $n = 2^k$.

Beispiel: Integermultiplikation

Ein rekursiver Ansatz:

Angenommen die Integer A und B haben Länge $n = 2^k$.



Beispiel: Integermultiplikation

Ein rekursiver Ansatz:

Angenommen die Integer A und B haben Länge $n = 2^k$.

b_{n-1} $\cdots$ b_0 $\times$ a_{n-1} $\cdots$ a_0

Beispiel: Integermultiplikation

Ein rekursiver Ansatz:

Angenommen die Integer A und B haben Länge $n = 2^k$.

b_{n-1}	$\cdots$	$b_{\frac{n}{2}}$	$b_{\frac{n}{2}-1}$	$\cdots$	b_0	$\times$	a_{n-1}	$\cdots$	$a_{\frac{n}{2}}$	$a_{\frac{n}{2}-1}$	$\cdots$	a_0

Beispiel: Integermultiplikation

Ein rekursiver Ansatz:

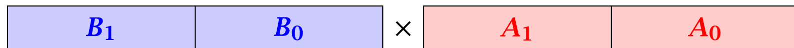
Angenommen die Integer A und B haben Länge $n = 2^k$.

B_1	B_0	$\times$	A_1	A_0

Beispiel: Integermultiplikation

Ein rekursiver Ansatz:

Angenommen die Integer A und B haben Länge $n = 2^k$.



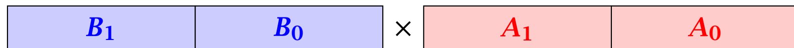
Dann gilt

$$A = A_1 \cdot 2^{\frac{n}{2}} + A_0 \text{ und } B = B_1 \cdot 2^{\frac{n}{2}} + B_0$$

Beispiel: Integermultiplikation

Ein rekursiver Ansatz:

Angenommen die Integer A und B haben Länge $n = 2^k$.



Dann gilt

$$A = A_1 \cdot 2^{\frac{n}{2}} + A_0 \text{ und } B = B_1 \cdot 2^{\frac{n}{2}} + B_0$$

Also,

$$A \cdot B = A_1 B_1 \cdot 2^n + (A_1 B_0 + A_0 B_1) \cdot 2^{\frac{n}{2}} + A_0 B_0$$

Beispiel: Integermultiplikation

```
1  Input:  Zahlen A, B, repraesentiert durch Bitarrays
2          der Laenge n
3  Output: Bitarray, das A*B enthaelt
4
5  mult(A, B, n)
6      if (n == 1)
7          return new int(A[0]*B[0]);
8      split(A,A0,A1);
9      split(B,B0,B1);
10     Z2 = mult(A1,B1,n/2);
11     Z1 = mult(A0,B1,n/2) + mult(A1,B0,n/2);
12     Z0 = mult(A0,B0,n/2);
13     return Z2*2^n + Z1*2^(n/2) + Z0;
```

Die Addition + addiert Bitarrays. Das funktioniert in C++ nicht direkt, aber man kann stattdessen z.B. eine Funktion add(A,B) benutzen. Wenn man den Algorithmus nach C++ übertragen möchte ist das Speichermanagement etwas unübersichtlich, da man jedes Zwischenergebnis vor Funktionsende wieder von Hand löschen muss.

Wir erhalten folgende Rekurrenzgleichung:

$$T(n) = 4T\left(\frac{n}{2}\right) + \mathcal{O}(n) \ .$$

Beispiel: Integermultiplikation

Mastertheorem: Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Fall 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Fall 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Beispiel: Integermultiplikation

Mastertheorem: Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Fall 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\qquad T(n) = \Theta(n^{\log_b a})$
- Fall 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\qquad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

In unserem Fall $a = 4$, $b = 2$, und $f(n) = \Theta(n)$. Also, Fall 1, da $n = \mathcal{O}(n^{2-\epsilon}) = \mathcal{O}(n^{\log_b a - \epsilon})$.

Beispiel: Integermultiplikation

Mastertheorem: Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Fall 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\qquad T(n) = \Theta(n^{\log_b a})$
- Fall 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\qquad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

In unserem Fall $a = 4$, $b = 2$, und $f(n) = \Theta(n)$. Also, Fall 1, da $n = \mathcal{O}(n^{2-\epsilon}) = \mathcal{O}(n^{\log_b a - \epsilon})$.

Wir erhalten Laufzeit $\mathcal{O}(n^2)$.

Beispiel: Integermultiplikation

Mastertheorem: Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Fall 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\qquad T(n) = \Theta(n^{\log_b a})$
- Fall 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\qquad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

In unserem Fall $a = 4$, $b = 2$, und $f(n) = \Theta(n)$. Also, Fall 1, da $n = \mathcal{O}(n^{2-\epsilon}) = \mathcal{O}(n^{\log_b a - \epsilon})$.

Wir erhalten Laufzeit $\mathcal{O}(n^2)$.

$\Longrightarrow$ Nicht besser als „Schulmethode“.

Beispiel: Integermultiplikation

We can use the following identity to compute Z_1:

Beispiel: Integermultiplikation

We can use the following identity to compute Z_1:

$$Z_1 = A_1B_0 + A_0B_1$$

Beispiel: Integermultiplikation

We can use the following identity to compute Z_1:

$$\begin{aligned} Z_1 &= A_1 B_0 + A_0 B_1 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - A_1 B_1 - A_0 B_0 \end{aligned}$$

Beispiel: Integermultiplikation

We can use the following identity to compute Z_1:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - \overbrace{A_1B_1}^{= Z_2} - \overbrace{A_0B_0}^{= Z_0} \end{aligned}$$

Beispiel: Integermultiplikation

We can use the following identity to compute Z_1:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - \overbrace{A_1B_1}^{=Z_2} - \overbrace{A_0B_0}^{=Z_0} \end{aligned}$$

```
1  Input:  Zahlen A, B, repraesentiert durch Bitarrays
2          der Laenge n
3  Output: Bitarray, das A*B enthaelt
4
5  mult(A, B, n)
6      if (n == 1)
7          return new int(A[0]*B[0]);
8      split(A,A0,A1);
9      split(B,B0,B1);
10     Z2 = mult(A1,B1,n/2);
11     Z1 = mult(A0+A1,B0+B1,n/2)-Z0-Z2;
12     Z0 = mult(A0,B0,n/2);
13     return Z2*2^n + Z1*2^(n/2) + Z0;
```

Beispiel: Integermultiplikation

Zeile 11 ist leider nicht korrekt, da A0 + A1 bzw. B0 + B1 eventuell $n/2+1$ bits haben können. Wenn man eine $n/2+1$-bit Zahl X in das höchstwertige Bit $X_{\frac{n}{2}}$ und die restlichen Bits $\bar{X}$ zerlegt kann man folgendes nutzen:

```
...
 X = A0 + A1;
 Y = B0 + B1;
Z1 = X_{n/2}*Y_{n/2}*2^n + (X_{n/2}*Ȳ + Y_{n/2}*X̄)*2^{n/2} + mult(X̄,Ȳ) - Z0 - Z2;
...
```

Laufzeit hierfür ist $T(n/2) + \mathcal{O}(n)$.

Beispiel: Integermultiplikation

Wir erhalten folgende Rekurrenz:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \ .$$

Master Theorem: Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Wir sind im Fall 1. Laufzeit $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$.

Eine deutliche Verbesserung der „Schulmethode".

Beispiel: Integermultiplikation

Wir erhalten folgende Rekurrenz:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \ .$$

Master Theorem: Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Wir sind im Fall 1. Laufzeit $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$.

Eine deutliche Verbesserung der „Schulmethode“.

Beispiel: Integermultiplikation

Wir erhalten folgende Rekurrenz:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \ .$$

Master Theorem: Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Wir sind im Fall 1. Laufzeit $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$.

Eine deutliche Verbesserung der „Schulmethode“.

Beispiel: Integermultiplikation

Wir erhalten folgende Rekurrenz:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \ .$$

Master Theorem: Rekurrenz: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Wir sind im Fall 1. Laufzeit $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$.

Eine deutliche Verbesserung der „Schulmethode“.