Übung zu
Algorithmen und Datenstrukturen (für ET/IT)

Sommersemester 2017

Rüdiger Göbl, Mai Bui

Computer Aided Medical Procedures
Technische Universität München

Wiederholung: Validation

- Validation: nicht-formaler Nachweis der Korrektheit, etwa durch systematisches Testen

Warum Validation wenn wir Verifikation haben?

- bei Verifikation können Fehler unterlaufen sein
- Verifikation zu aufwendig oder nicht mehr möglich für größere bzw. komplexe Programme
- der verwendete Compiler/Rechner kann fehlerhaft sein

Wiederholung: Validation

- Validation: nicht-formaler Nachweis der Korrektheit, etwa durch systematisches Testen

Warum Validation wenn wir Verifikation haben?

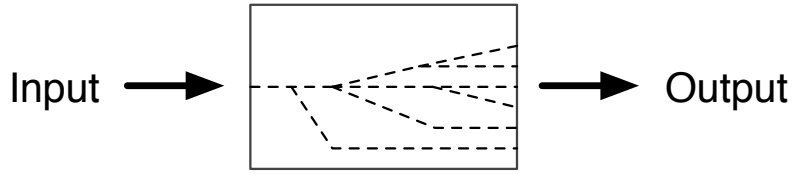
- bei Verifikation können Fehler unterlaufen sein
- Verifikation zu aufwendig oder nicht mehr möglich für größere bzw. komplexe Programme
- der verwendete Compiler/Rechner kann fehlerhaft sein

- aber! Testen kann nur Anwesenheit von Fehlern zeigen, nicht die Abwesenheit!

Validation **Multiply**

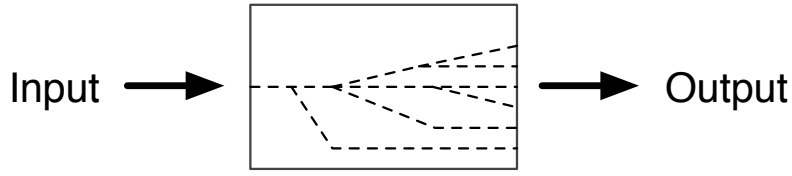
- Funktion **Multiply**(a,b) mit vorzeichenbehafteten (signed) Variablen von 8 Bit Länge
  $\Rightarrow$ insgesamt mögliche Zahlendarstellung von $2^8 = 256$
  $\Rightarrow$ Wertebereich von $[-2^7, 2^7 - 1]$
- Extremfälle an Grenzwerten -2^7 und $2^7 - 1$
  $\Rightarrow$ z.B. Überlauf
- Extremfall bei 0
  $\Rightarrow$ Übergang von negativen auf positiven Zahlen

Wiederholung: Whitebox-Test



- Programm als “Whitebox” betrachten
  - innere Struktur bekannt
- systematischer Test auf innere Struktur des Programms, d.h. das alle Programmteile ausgeführt werden $\rightarrow$ ablaufbezogenes Testen

Wiederholung: Whitebox-Test



- Programm als "Whitebox" betrachten
  - innere Struktur bekannt
- systematischer Test auf innere Struktur des Programms, d.h. das alle Programmteile ausgeführt werden → ablaufbezogenes Testen
- Test-Varianten (Auszug):
  - Ausführung sämtlicher Programmwege inkl. Kombinationen (meist unpraktikabel!)
  - Alle Schleifen müssen nicht nur einmal, sondern zweimal durchlaufen werden
  - Alle Anweisungen sollen mindestens einmal ausgeführt werden

White-Box-Test **Multiply**

```
Multiply(a, b):
    if (a == 0  ∨  b == 0) {
            c = 0;
    } else if (a < 0) {
            c = −Multiply(−a, b);
    } else {
            c = 0;
            i = 1;
            while(i ≤ a) {
                    c = c + b;
                    i = i + 1;
            }
    }
    return c;
```

- Fall 1: $a = 0, b$ beliebig oder $b = 0, a$ beliebig

White-Box-Test **Multiply**

```
Multiply(a, b):
        if (a == 0  ∨  b == 0) {
                c = 0;
        } else if (a < 0) {
                c = −Multiply(−a, b);
        } else {
                c = 0;
                i = 1;
                while(i ≤ a) {
                        c = c + b;
                        i = i + 1;
                }
        }
        return c;
```

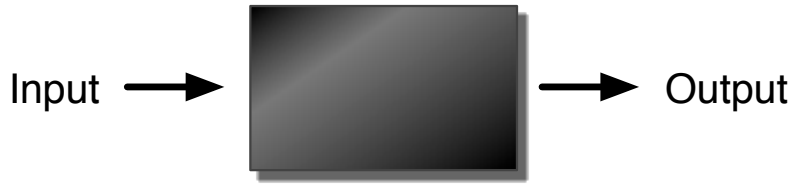
- Fall 2: $a < 0, b \neq 0$

White-Box-Test **Multiply**

```
Multiply(a, b):
        if (a == 0  ∨  b == 0) {
                c = 0;
        } else if (a < 0) {
                c = −Multiply(−a, b);
        } else {
                c = 0;
                i = 1;
                while(i ≤ a) {
                        c = c + b;
                        i = i + 1;
                }
        }
        return c;
```

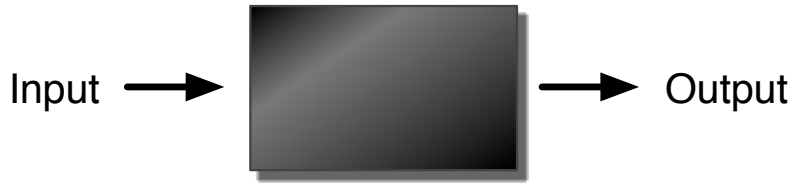
- Fall 3: $a > 0, b \neq 0, a \geq 2$ für doppelten Durchlauf der Schleife

Wiederholung: Blackbox-Test



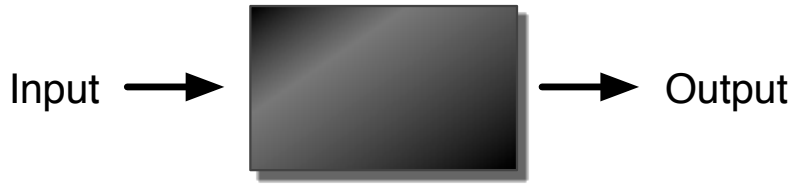
- Programm als “Blackbox” betrachten
  - interne Struktur unbekannt
  - nur Eingabe und Ausgabe bekannt
  - sowie Dokumentation was als Eingabe erlaubt ist

Wiederholung: Blackbox-Test



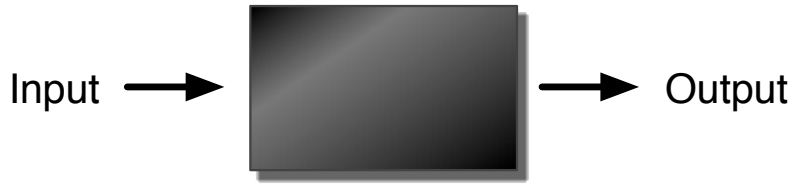
- Programm als “Blackbox” betrachten
  - interne Struktur unbekannt
  - nur Eingabe und Ausgabe bekannt
  - sowie Dokumentation was als Eingabe erlaubt ist
- systematischer Test auf mögliche Eingabedaten (und ob korrektes Ergebnis geliefert wird) $\rightarrow$ datenbezogenes Testen

Wiederholung: Blackbox-Test



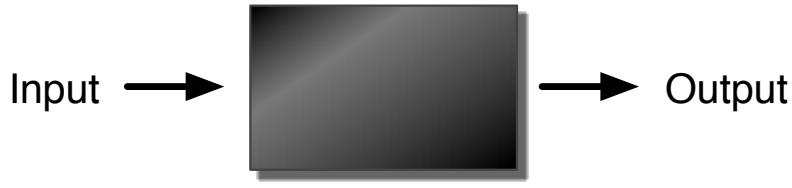
- Programm als "Blackbox" betrachten
  - interne Struktur unbekannt
  - nur Eingabe und Ausgabe bekannt
  - sowie Dokumentation was als Eingabe erlaubt ist
- systematischer Test auf mögliche Eingabedaten (und ob korrektes Ergebnis geliefert wird) $\rightarrow$ datenbezogenes Testen
- Repräsentative Werteanalyse:
  - Klassen von Eingabedaten (z.B. negative ganze Zahlen)
  - Auswahl von Repräsentanten dieser Klasse zum Test

Wiederholung: Blackbox-Test



- Programm als "Blackbox" betrachten
  - interne Struktur unbekannt
  - nur Eingabe und Ausgabe bekannt
  - sowie Dokumentation was als Eingabe erlaubt ist
- systematischer Test auf mögliche Eingabedaten (und ob korrektes Ergebnis geliefert wird) $\rightarrow$ datenbezogenes Testen
- Repräsentative Werteanalyse:
  - Klassen von Eingabedaten (z.B. negative ganze Zahlen)
  - Auswahl von Repräsentanten dieser Klasse zum Test
- Grenzwertanalyse:
  - Eingabedaten haben meist Grenzwerte (z.B. Zahlen von -128 bis 127)
  - direkte Tests auf die Grenzwerte und auch darüber hinaus

Black-Box-Test



- Programm als "BlackBox" betrachten
  - interne Struktur unbekannt
  - nur Eingabe und Ausgabe bekannt
  - sowie Dokumentation was als Eingabe erlaubt ist

Gegeben ein Sortieralgorithmus:
$\Rightarrow$ nur **Sortier-Eigenschaft** und **Stabilität** testbar
$\Rightarrow$ keine Information über den Algorithmus

Komplexität von Algorithmen

Interessante Fragen bei Algorithmus A:

- wie viel Speicherplatz benötigt Algorithmus?
- wie viel Rechenzeit benötigt Algorithmus?

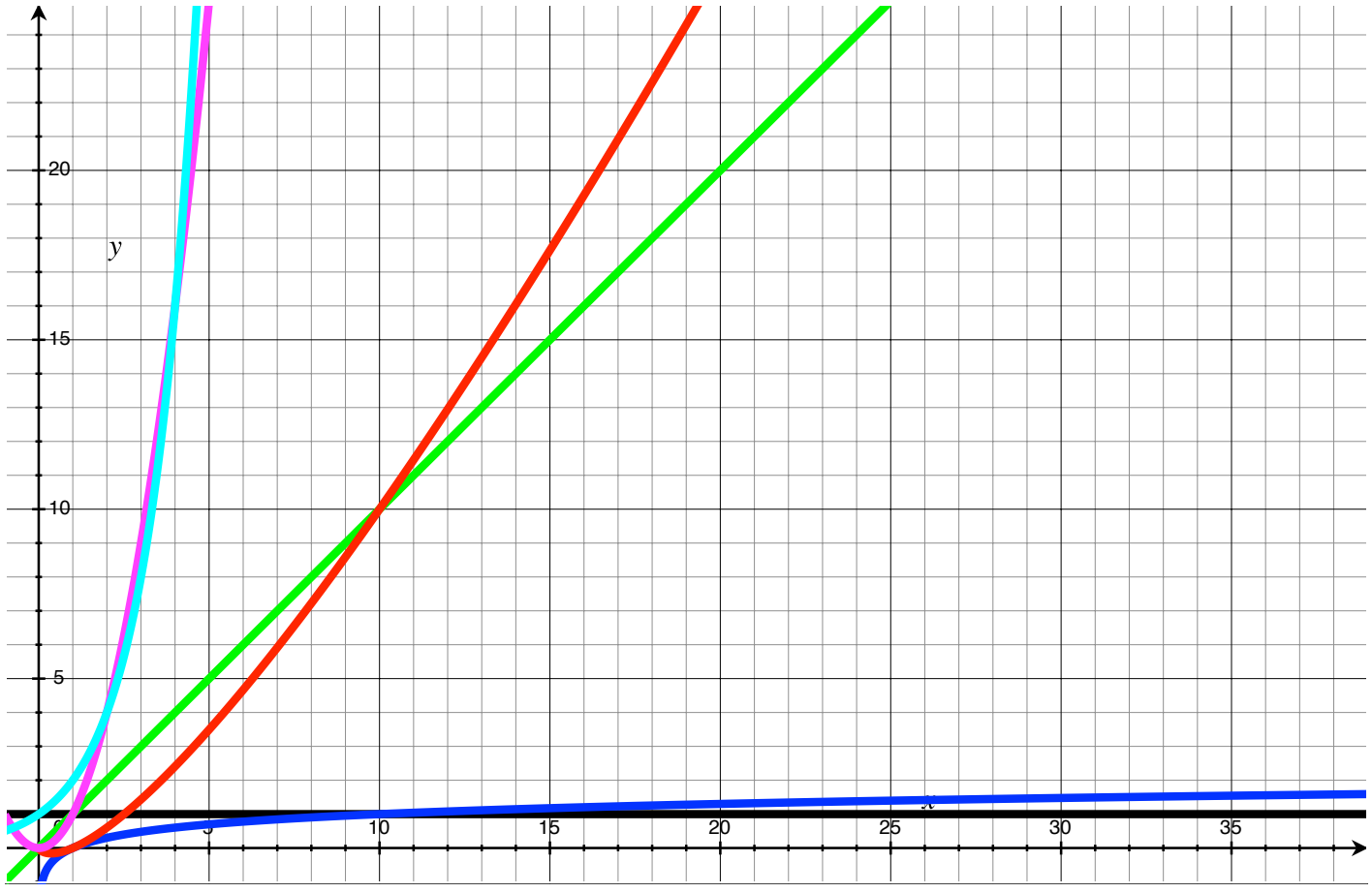
→ Komplexitätsanalyse

Wesentlicher Faktor bei Komplexität: Größe der Eingabe n

- benötigter Speicher abhängig von n
- benötigte Rechenzeit abhängig von n

→ Funktion T zur Beschreibung des Wachstumsverhaltens von A

Wachstumsraten



schwarz: $f(n) = 1$ blau: $f(n) = \log(n)$ grün: $f(n) = n$

rot: $f(n) = n \log(n)$ pink: $f(n) = n^2$ cyan: $f(n) = 2^n$

Landau-Symbole

Abschätzung des Wachstumsverhaltens einer (komplexeren) Funktion durch eine andere (intuitivere) Funktion.

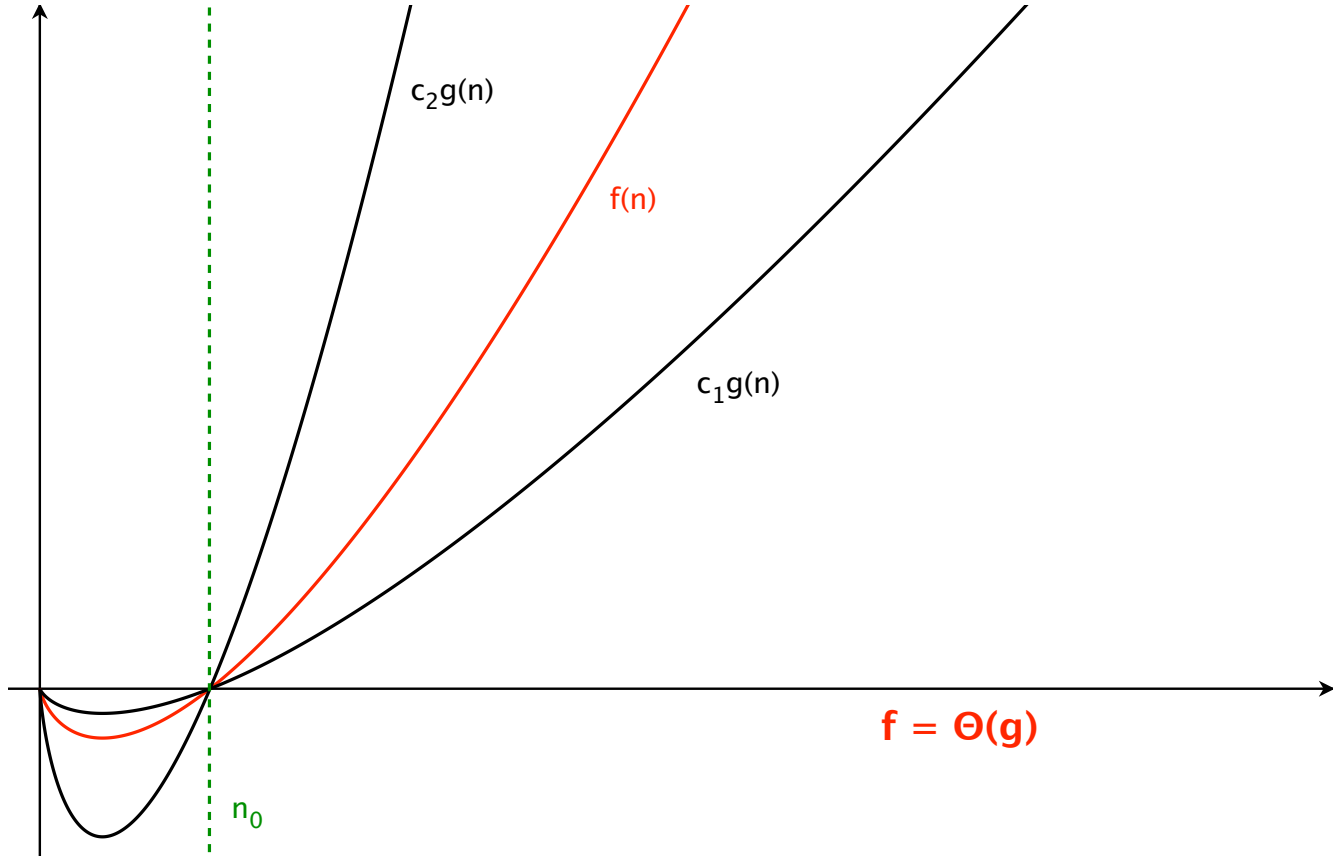
$$\begin{aligned}\Theta(g) \quad := \quad \{f : \mathbb{R} \to \mathbb{R} \ : \ &\text{es existieren } c_1, c_2 > 0 \text{ und } n_0 \in \mathbb{N}, \\ &\text{so dass für alle } n \geq n_0 \text{ gilt:} \\ &0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}\end{aligned}$$

$$\begin{aligned}O(g) \quad := \quad \{f : \mathbb{R} \to \mathbb{R} \ : \ &\text{es existieren } c > 0 \text{ und } n_0 \in \mathbb{N}, \\ &\text{so dass für alle } n \geq n_0 \text{ gilt:} \\ &0 \leq f(n) \leq c \cdot g(n)\}\end{aligned}$$

$\Theta(g)$ ist die *asymptotisch scharfe* und $O(g)$ die *asymptotisch obere Schranke*.

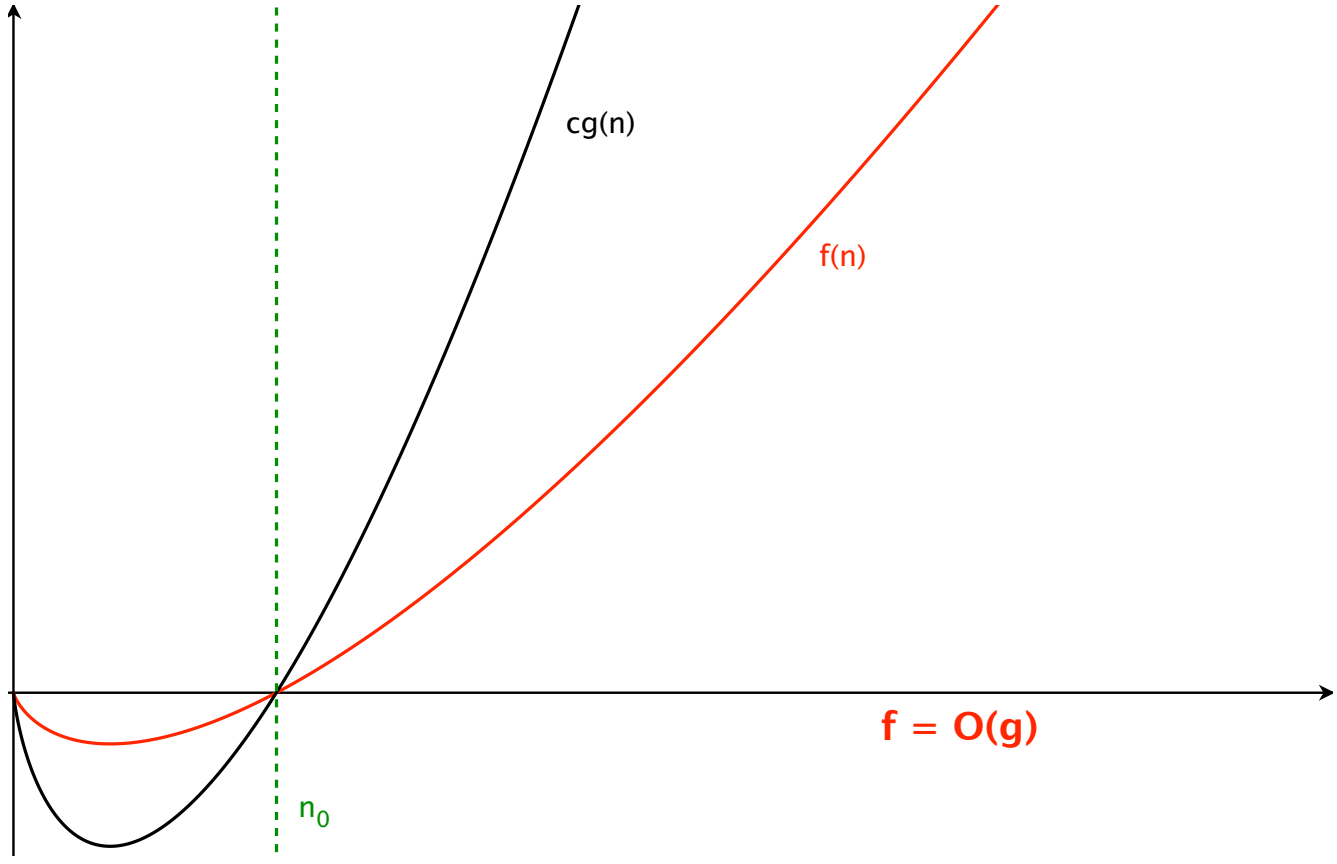
Landau-Symbol Θ

$$\Theta(g) = \{ f : \mathbb{R} \to \mathbb{R} : \text{ es existieren } c_1, c_2 > 0, n_0 \in \mathbb{N} \text{ so dass für alle } n \geq n_0 \text{ gilt: } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}$$



Landau-Symbol O

$$O(g) = \{f : \mathbb{R} \to \mathbb{R} : \text{ es existieren } c > 0 \text{ und } n_0 \in \mathbb{N} \text{ so dass}$$
$$\text{für alle } n \geq n_0 \text{ gilt: } 0 \leq f(n) \leq cg(n)\}$$



$7n^4 = O(n^5)$

Betrachte Ungleichung aus der Definition von $O(n)$:

$$0 \leq 7n^4 \leq c \cdot n^5$$

Dividiere für $n \geq 1$ durch n^4 ohne Umkehr der Ungleichung:

$$0 \leq 7 \leq c \cdot n$$

Für $c = 7$ und $n_0 = 1$, so gilt die Ungleichung für alle $n \geq n_0$. □

$n^2/2 - 2n = \Theta(n^2)$

Betrachte beide Schranken unabhängig voneinander:

$$c_1 \cdot n^2 \leq \frac{n^2}{2} - 2n \qquad \frac{n^2}{2} - 2n \leq c_2 \cdot n^2$$

Dividiere für $n \geq 1$ durch n^2 ohne Umkehr der Ungleichung:

$$c_1 \leq \frac{1}{2} - \frac{2}{n} \qquad \frac{1}{2} - \frac{2}{n} \leq c_2$$

Für $c_1 = \frac{1}{4}$, $c_2 = \frac{1}{2}$ und $n_0 = 8$ gilt die komplette Ungleichung für alle $n \geq n_0$. □

$2^{n+1} = O(2^n)$

Betrachte wieder die Ungleichung aus der Definition von $O(2^n)$:

$$0 \leq 2^{n+1} \leq c \cdot 2^n$$

Zerlegung ergibt:

$$2^{n+1} = 2 \cdot 2^n$$

Für $c = 2$ und $n_0 = 1$ gilt also die Ungleichung für alle $n \geq n_0$. $\square$

$2^{2n} \neq O(2^n)$

Annahme, dass die Behauptung **nicht** zutrifft. Dann existieren Konstanten c und $n_0 > 0$, so dass für alle $n \geq n_0$ gilt:

$$0 \leq 2^{2n} \leq c \cdot 2^n$$

Zerlegung ergibt:

$$2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n \Leftrightarrow 2^n \leq c \ .$$

Die Folge 2^n ist aber **unbeschränkt**, und es existiert keine Konstante c, die für beliebige n die Forderung $c \geq 2^n$ erfüllt.

Widerspruch zur Annahme!
Also gilt die ursprüngliche Behauptung. □

Bedeutung der konstanten Faktoren

Die Worst-Case-Komplexität eines Algorithmus sei $\Theta(n)$. Nachdem auch $2n = \Theta(n)$, entspricht die Dauer eines Worst-Case-Beispiels der Länge $2n$ ungefähr der Dauer eines Worst-Case-Beispiels der Länge n.

- Konstante Faktoren entfallen, nur grundsätzliches Wachstum ist relevant.
- Laufzeitunterschied von konstanten Faktoren also unabhänging von n.
- Doppelt großes Beispiel rechnet auch (etwa) doppelt so lange!

Die Behauptung ist **falsch**! □

Iterationen über Code bekannter Komplexität

Sei die Komplexität einer Funktion f bestimmt als $O(n)$. Dann ist die n-malige Ausführung $O(n^2)$.

- $T_f(n) = O(n)$, also gibt es c und n_0, so dass für alle $n \geq n_0$:

$$0 \leq T_f(n) \leq c \cdot n$$

- Bei n-maliger Ausführung muss einfach multipliziert werden:

$$0 \leq n \cdot T_f(n) \leq n \cdot c \cdot n = c \cdot n^2$$

Das ist direkt die Forderung aus der Definition von $O(n^2)$, die Aussage ist **korrekt**! □

Falls $f(n) = \Theta(g(n))$, dann folgt $2^{f(n)} = \Theta(2^{g(n)})$.

- Für $f(n) = n$ und $g(n) = 2 \cdot n$ gilt $f(n) = \Theta(g(n))$.
- Falls Aussage korrekt, erhalten wir aus $2^{f(n)} = \Theta(2^{g(n)})$:

$$0 \leq c_1 \cdot 2^{2 \cdot n} \leq 2^n \leq c_2 \cdot 2^{2 \cdot n}$$

- Teilung durch $2^n > 0$ $(n > 0)$ ergibt:

$$0 \leq c_1 \cdot 2^n \leq 1 \leq c_2 \cdot 2^n$$

- **Widerspruch**, da Folge 2^n unbeschränkt ist, also kein konstantes c_1 gefunden werden kann.

Damit ist die komplette Aussage **falsch**, da mit der getroffenen Auswahl für f und g ein Gegenbeispiel gefunden wurde. □

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

- Falls die Aussage wahr ist, dann gilt:

$$0 \leq n^n \leq c \cdot 2^n$$

- Teilung durch $2^n > 0$ ($n > 0$) ergibt:

$$\left(\frac{n}{2}\right)^n \leq c$$

- **Widerspruch**, da die Folge unbeschränkt ist. Es existiert keine obere Schranke c.

Damit ist die Aussage als **falsch** widerlegt! □

$n = O(n^3)$

- Falls die Aussage wahr ist, dann gilt:

$$0 \leq n \leq c \cdot n^3$$

- Teilung durch $n^3 > 0$ ($n > 0$) ergibt:

$$\left(\frac{1}{n^2}\right) \leq c$$

- $\left(\frac{1}{n^2}\right) \to 0$, also ist $\left(\frac{1}{n^2}\right)$ beschränkt.
  Beispiel: Für $c = 1$ und $n_0 = 10$ für alle $n \geq n_0$

Damit ist die Aussage als **wahr**! □

Wiederholung: RAM-Modell

- einfaches Rechnermodell für Laufzeitanalyse
- Ziel: Abschätzung der erforderlichen Ressourcen zur Ausführung eines Algorithmus
  - exakte Zeit der Ausführung ist nicht relevant
- Modell: Random Access Machine, kurz RAM

Wiederholung: Annahmen im RAM-Modell

- nur sequentielle Ausführungen
  - das heißt nur 1 Prozessor, keine Parallelität
- alle Daten liegen direkt zugreifbar im Speicher ($\rightarrow$ RAM)
- jeder Speicherzugriff dauert gleich lang
  - das ist in Wahrheit nicht der Fall! (Speicher-Hierarchie)
- alle elementaren Verarbeitungsschritte benötigen eine Zeiteinheit
  - Wertzuweisung
  - Arithmetische Operationen: $+$, -, *, $/$, %, ceil, floor
  - Vergleichsoperationen: $<$, $>$, $!=$
  - Kontrollflußoperationen: **if**, **else**

Komplexität von **Swap**

Swap(A, i, j)**:**

$k = A[j];$
$A[j] = A[i];$
$A[i] = k;$

- Im RAM-Modell 3 Anweisungen
- Damit offensichtlich $O(1)$ (mit $c = 3$, $n_0 = 1$)

Komplexität eines Algorithmus

Algorithmus besteht aus elementaren Bausteinen

$\rightarrow$ Komplexität eines ganzen Algorithmus ergibt sich aus

- Komplexität der elementaren Bausteine
- und der Addition in O-Notation

Komplexität von **IndexOfMin**

```
1    i = j;
2    k = j + 1;
3    while (k < n) {
4        if (A[k] < A[i]) {
5            i = k;
         }
6        k = k + 1;
     }
```

Zeile	Kosten	Häufigkeit
1	c_1	1
2	c_2	1
3	c_3	$n-j$
4	c_4	$n-j-1$
5	c_5	t_k
6	c_6	$n-j-1$

- Laufzeitfunktion

$$\begin{aligned} T(n) &= 1+1+(n-j)+(n-j-1)+t_k+(n-j-1) \\ &= 3\cdot n-3\cdot j+t_k \end{aligned}$$

- t_k ist die Anzahl der Verzweigungen bei 4 bzw. Durchführungen von 5

Best und Worst Case der **IndexOfMin**

- Im *besten Fall* wird Zeile 5 nie ausgeführt. Das passiert, wenn A bereits *aufsteigend* sortiert ist.
- Im *schlechtesten Fall* wird Zeile 5 immer ausgeführt. Das passiert, wenn A bereits *absteigend* sortiert ist.

- Im schlechtesten Fall gilt $t_k = n - j - 1$, und damit

$$T(n) = 4 \cdot n - 4 \cdot j - 1$$

- Somit ergibt sich sogar dann die Komplexität $O(n)$!
- Inoffiziell ergibt sich das direkt aus der **for**-Schleife...

Komplexität von **SelectionSort**

```
1    for j = 0 to n − 2 {
2        i = IndexOfMin(A, j);
3        if (j ≠ i) {
4            Swap(A, i, j);
         }
     }
```

Zeile	Kosten	Häufigkeit
1	c_1	$2n$
2	c_2	$(n-1) \cdot O(n)$
3	c_3	$n-1$
4	c_4	$t_j \cdot O(1)$

- Laufzeitfunktion ungefähr

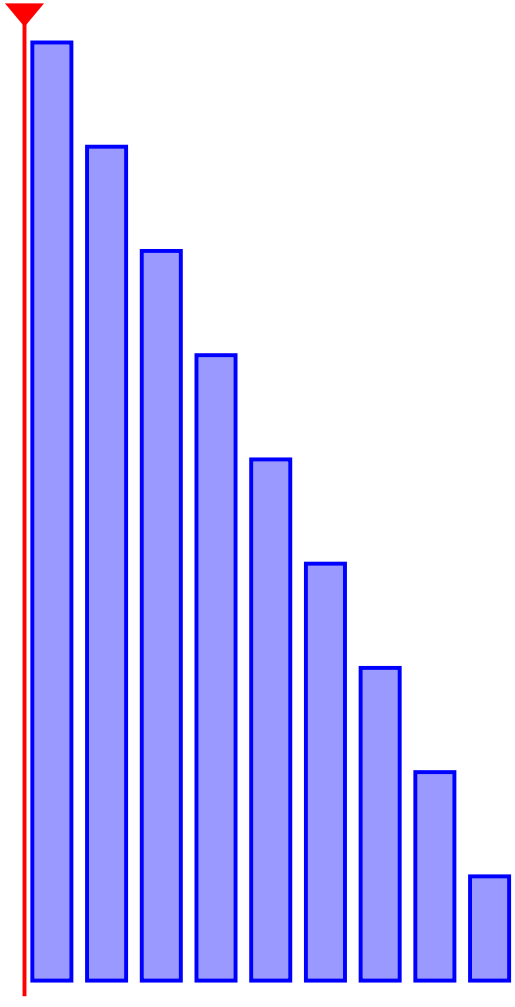
$$\begin{aligned} T(n) &= 2n + \boxed{(n-1) \cdot O(n)} + (n-1) + t_j \cdot O(1) \\ &= O(n^2) \end{aligned}$$

- t_j ist die Anzahl der Vertauschungen bei 4

Best und Worst Case der **SelectionSort**

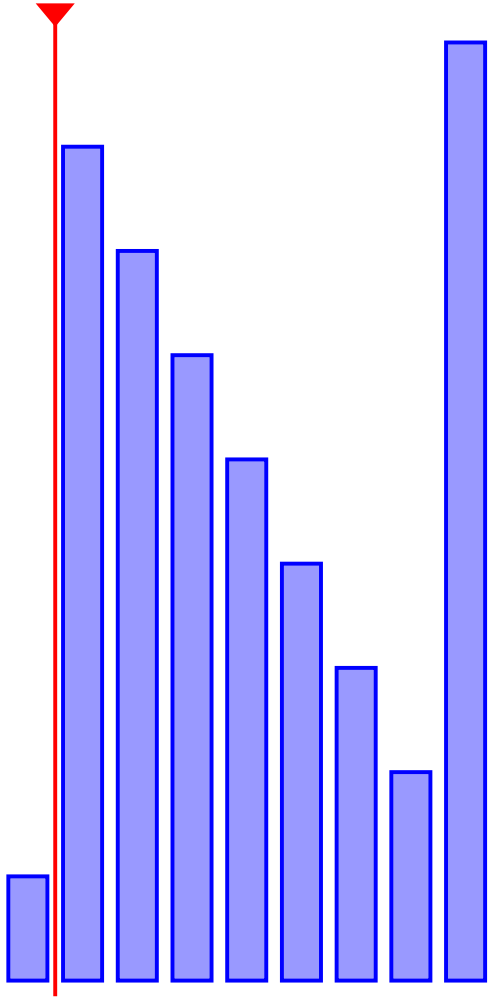
- Im *besten Fall* ist A bereits aufsteigend sortiert.
- Der *schlechteste Fall* der **IndexOfMin** war ein absteigend sortiertes A, aber dann genügen $n/2$ Vertauschungen.

Selection Sort: Absteigende Sortierung



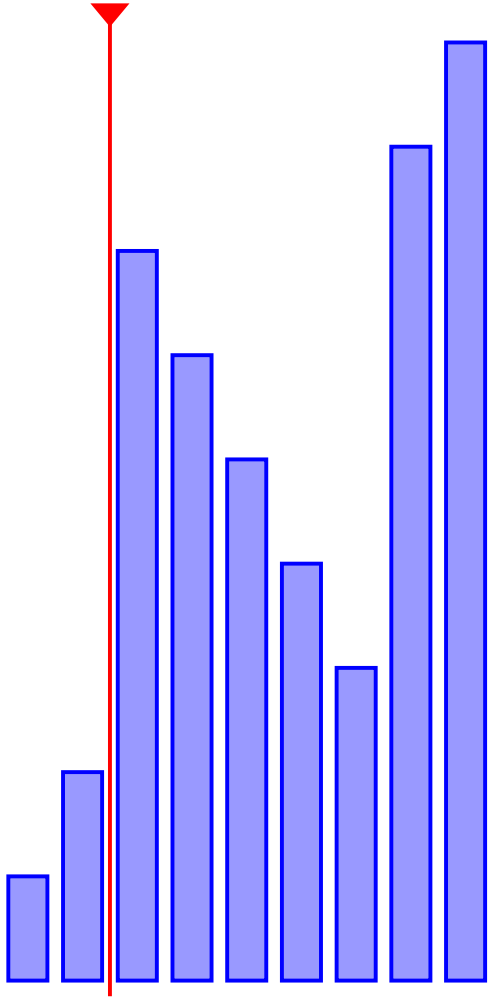
$j = 0$: 0 Updates, 0 Vertauschungen

Selection Sort: Absteigende Sortierung



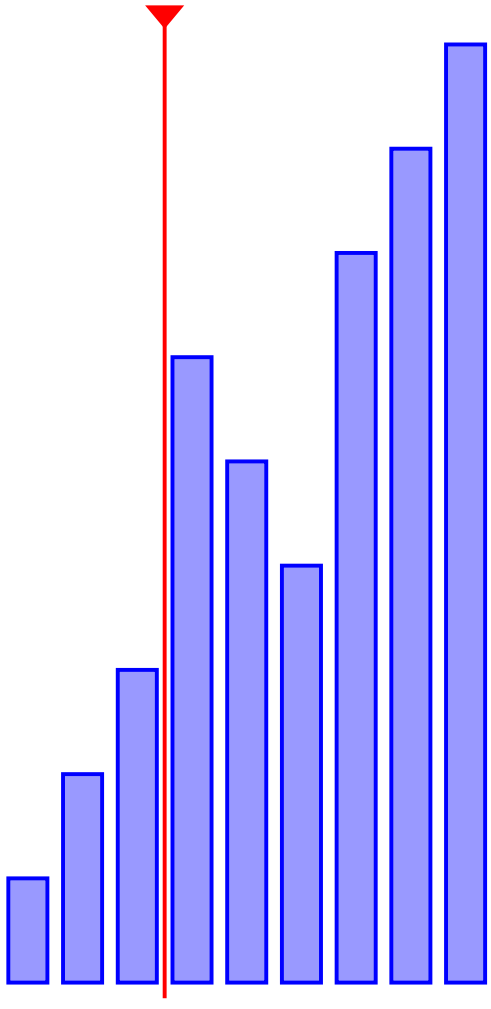
$j = 1$: 8 Updates, 1 Vertauschung

Selection Sort: Absteigende Sortierung



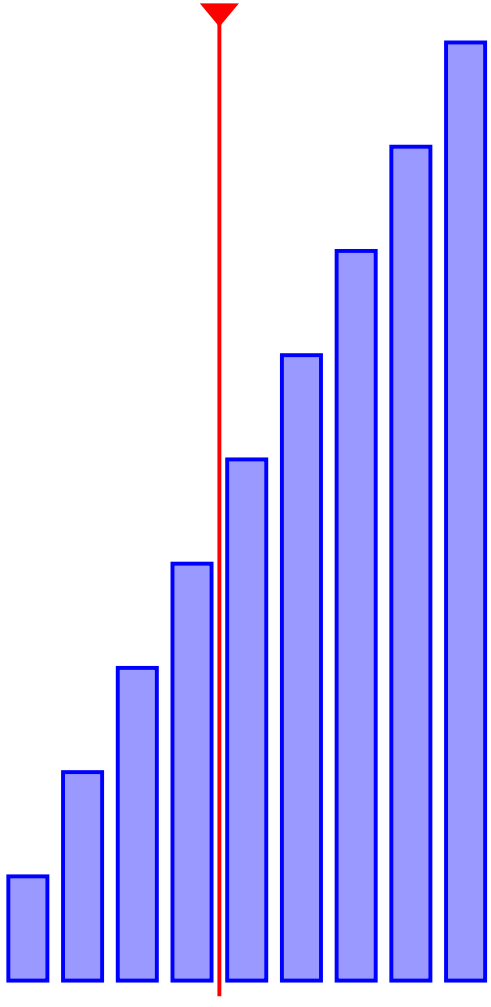
$j = 2$: 14 Updates, 2 Vertauschungen

Selection Sort: Absteigende Sortierung



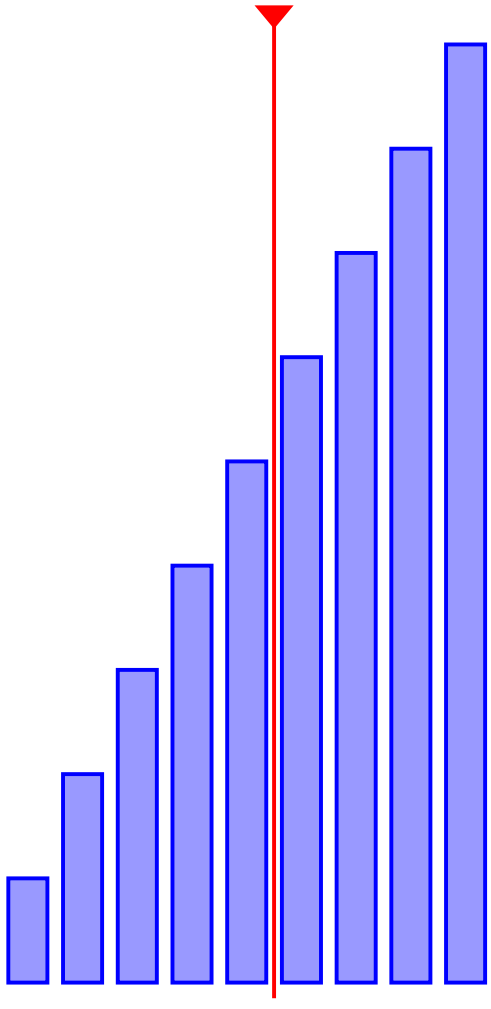
$j = 3$: 18 Updates, 3 Vertauschungen

Selection Sort: Absteigende Sortierung



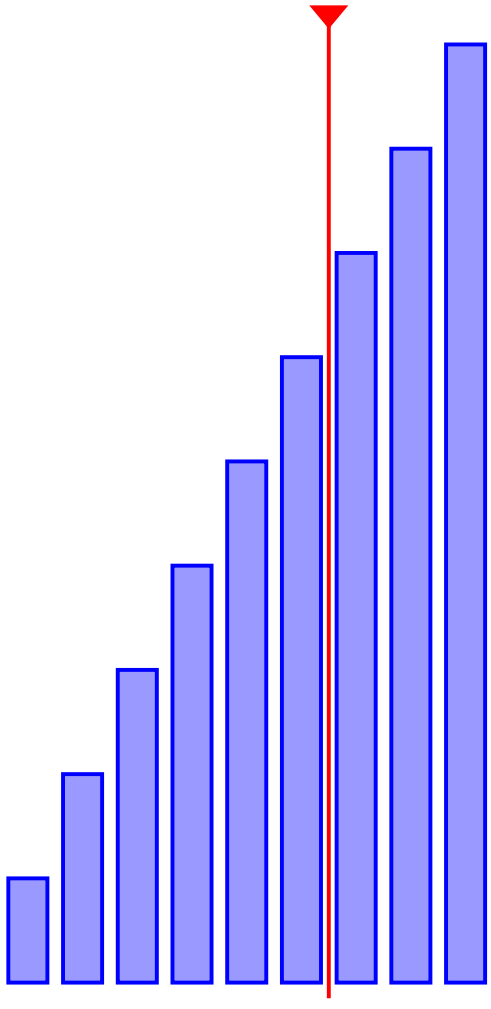
$j = 4$: 20 Updates, 4 Vertauschungen

Selection Sort: Absteigende Sortierung



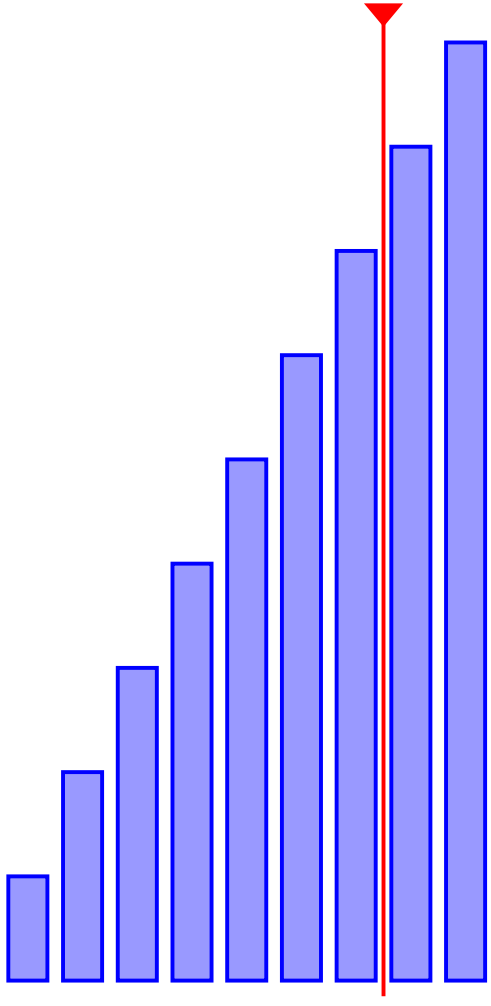
$j = 5$: 20 Updates, 4 Vertauschungen

Selection Sort: Absteigende Sortierung



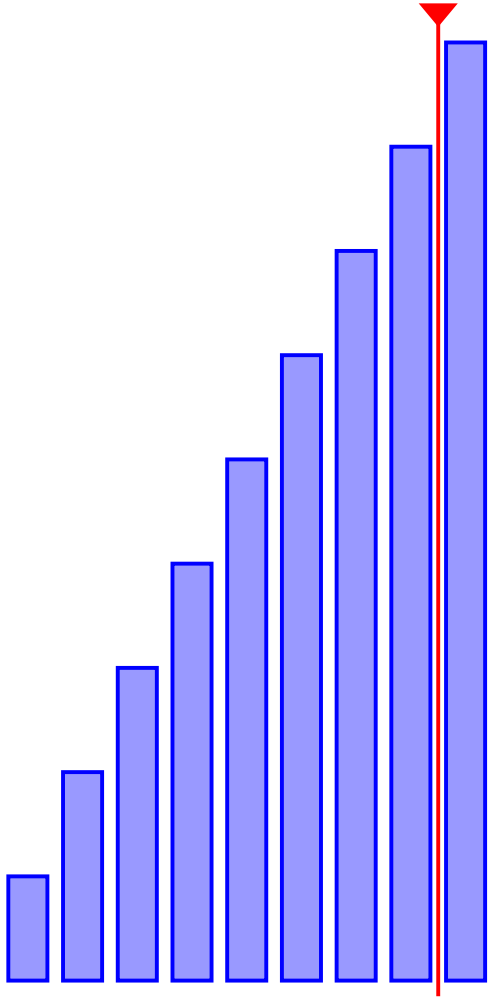
$j = 6$: 20 Updates, 4 Vertauschungen

Selection Sort: Absteigende Sortierung



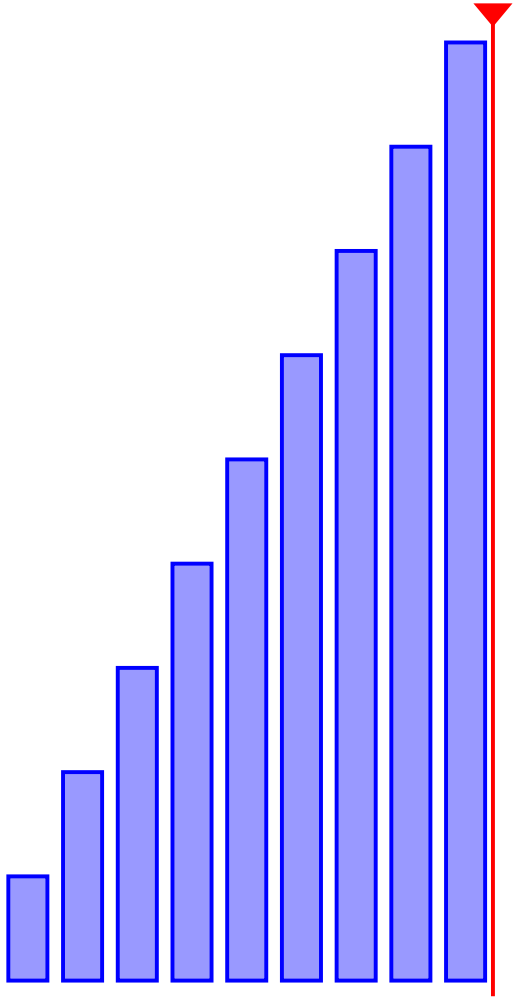
$j = 7$: 20 Updates, 4 Vertauschungen

Selection Sort: Absteigende Sortierung



$j = 8$: 20 Updates, 4 Vertauschungen

Selection Sort: Absteigende Sortierung



$j = 9$: 20 Updates, 4 Vertauschungen

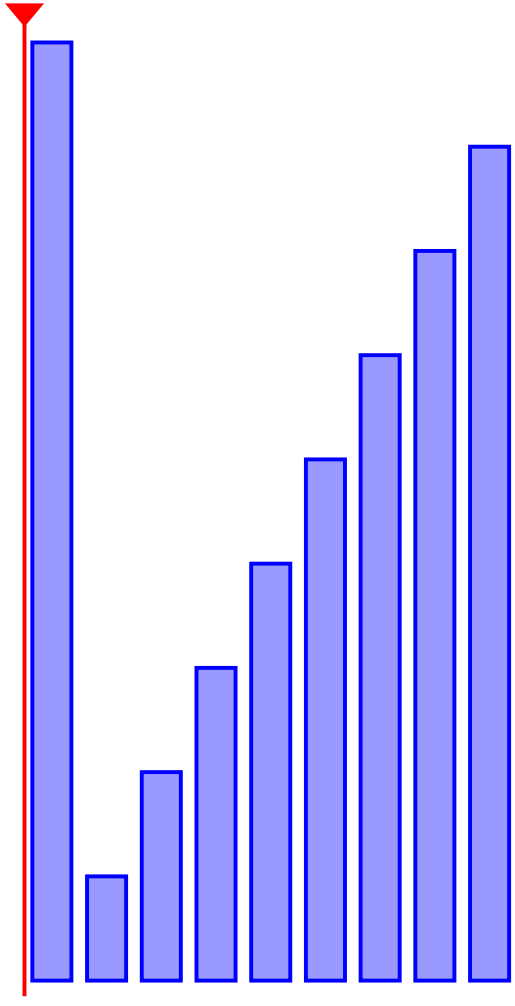
Best und Worst Case der **SelectionSort**

- Im *besten Fall* ist A bereits aufsteigend sortiert.
- Der *schlechteste Fall* der **IndexOfMin** war ein absteigend sortiertes A, aber dann genügen $n/2$ Vertauschungen.

Best und Worst Case der **SelectionSort**

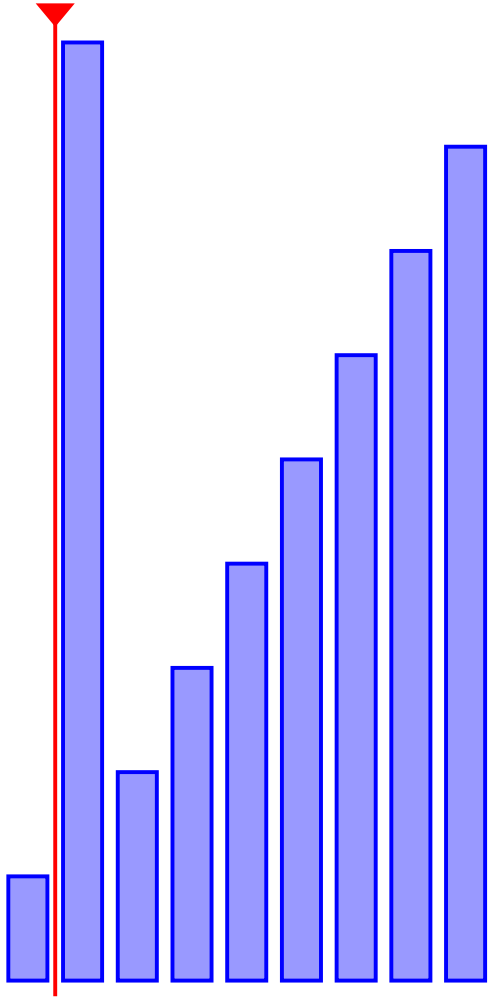
- Im *besten Fall* ist A bereits aufsteigend sortiert.
- Der *schlechteste Fall* der **IndexOfMin** war ein absteigend sortiertes A, aber dann genügen $n/2$ Vertauschungen.
- Maximale Anzahl von Vertauschungen, wenn A das maximale Element am Anfang und dann eine aufsteigende Ordnung hat.

Selection Sort: Größtes Element vorne, dann aufsteigend



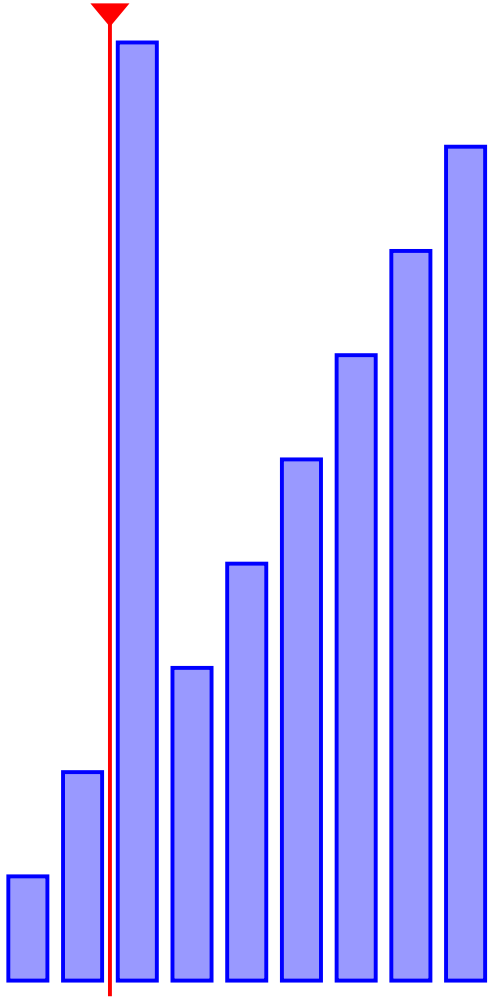
$j = 0$: 0 Updates, 0 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



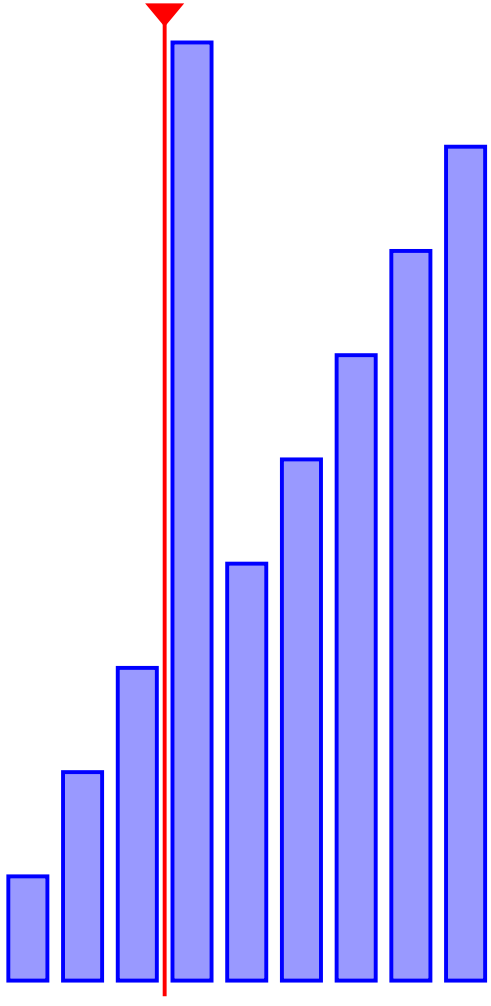
$j = 1$: 1 Update, 1 Vertauschung

Selection Sort: Größtes Element vorne, dann aufsteigend



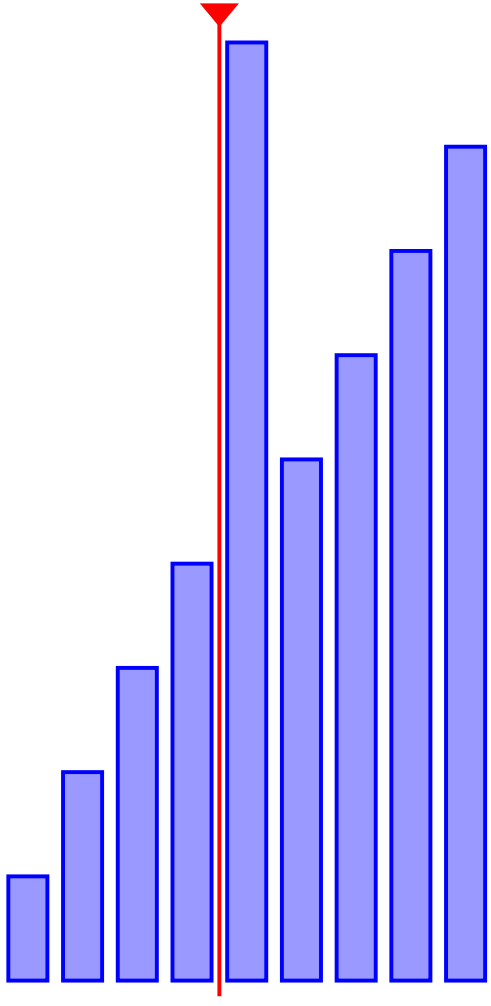
$j = 2$: 2 Updates, 2 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



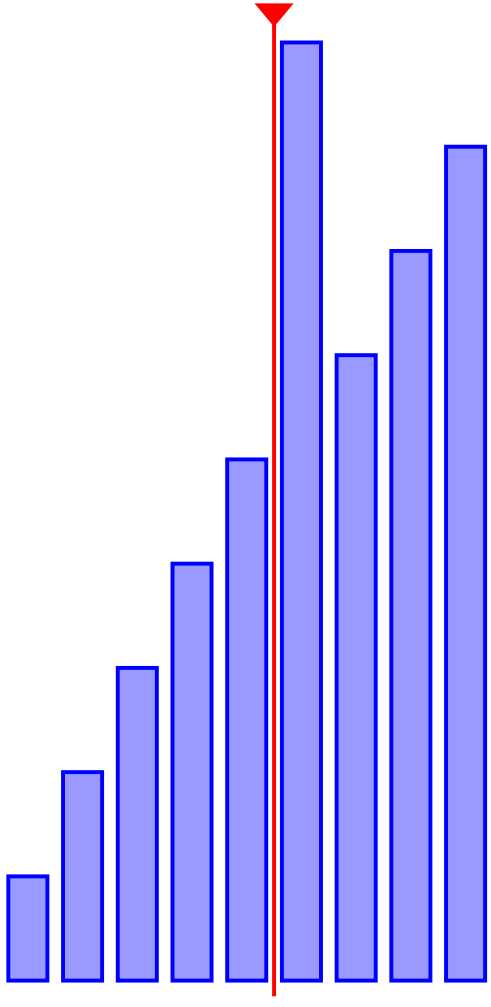
$j = 3$: 3 Updates, 3 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



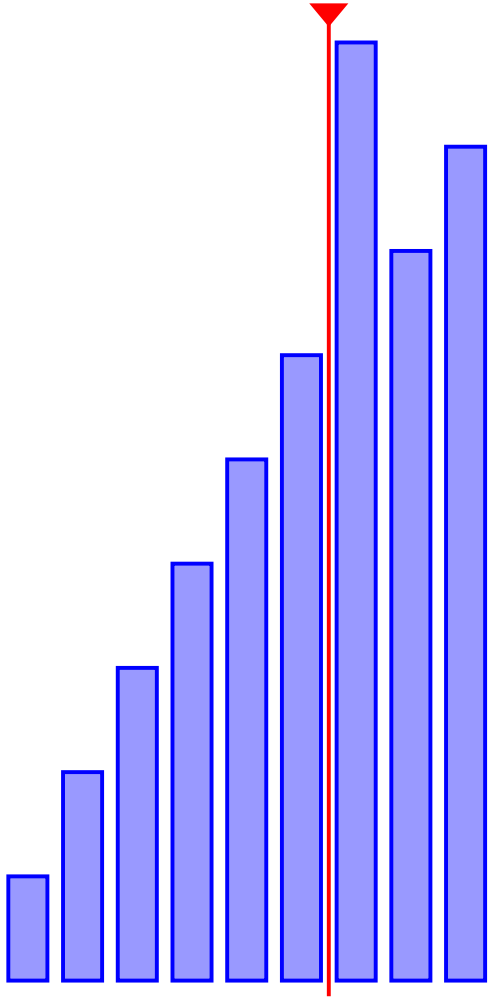
$j = 4$: 4 Updates, 4 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



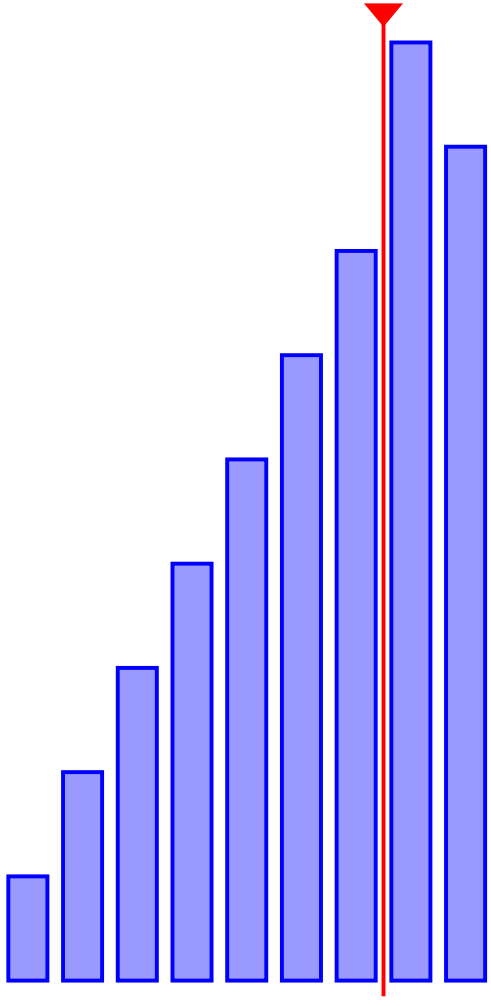
$j = 5$: 5 Updates, 5 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



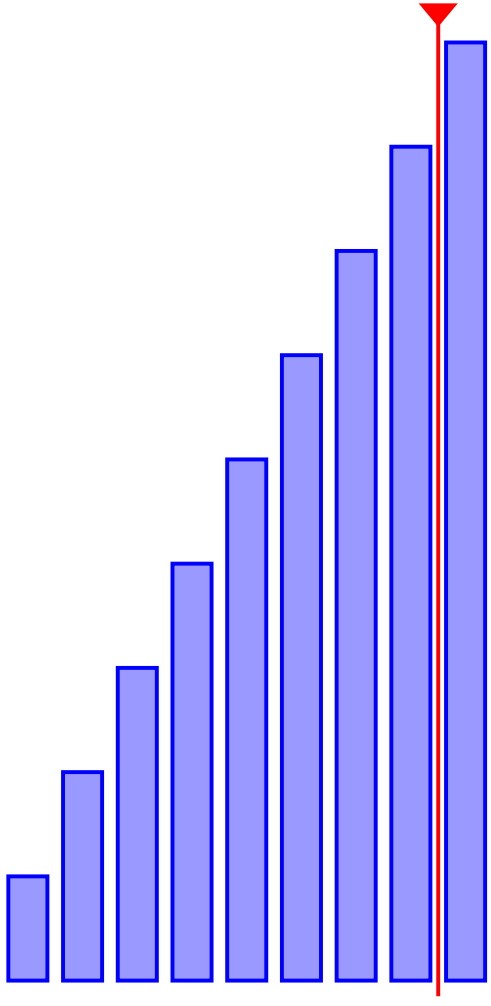
$j = 6$: 6 Updates, 6 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



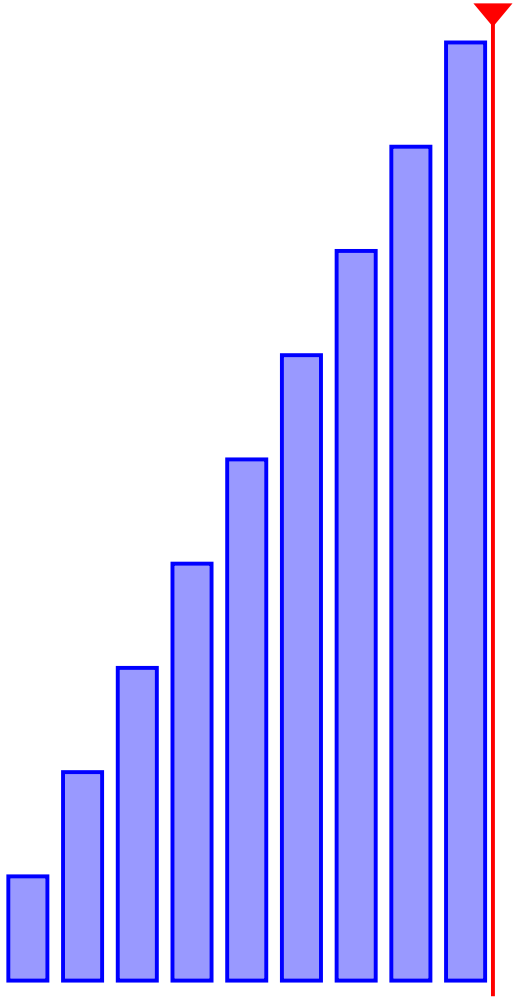
$j = 7$: 7 Updates, 7 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



$j = 8$: 8 Updates, 8 Vertauschungen

Selection Sort: Größtes Element vorne, dann aufsteigend



$j = 9$: 8 Updates, 8 Vertauschungen

Best und Worst Case der **SelectionSort**

- Im *besten Fall* ist A bereits aufsteigend sortiert.
- Der *schlechteste Fall* der **IndexOfMin** war ein absteigend sortiertes A, aber dann genügen $n/2$ Vertauschungen.
- Maximale Anzahl von Vertauschungen, wenn A das maximale Element am Anfang und dann eine aufsteigende Ordnung hat.

Best und Worst Case der **SelectionSort**

- Im *besten Fall* ist A bereits aufsteigend sortiert.
- Der *schlechteste Fall* der **IndexOfMin** war ein absteigend sortiertes A, aber dann genügen $n/2$ Vertauschungen.
- Maximale Anzahl von Vertauschungen, wenn A das maximale Element am Anfang und dann eine aufsteigende Ordnung hat.

- Sind Vertauschungen oder Updates des minimalen Index teurer?

Konstanten in Landau-Symbolen

- Landau-Symbole Θ, O enthalten Konstanten c_1, c_2 bzw. c

  $\rightarrow$ für asymptotisches Verhalten ($n \rightarrow \infty$) vernachlässigbar

- Aber: ist n klein, spielen die Konstanten sehr wohl eine Rolle!
  - es kann angebracht sein einen eigentlich komplexeren Algorithmus einzusetzen, da bei kleinem n die Konstanten mehr zählen
  - Beispiel: Insertion Sort

Effizienz von Algorithmen I

O-Notation erlaubt Kategorisierung der Effizienz von Algorithmen

- $O(1)$: konstante Laufzeit
  - unabhängig von Problemgröße
  - *Beispiel:* Löschen von erstem Element in verketteter Liste

Effizienz von Algorithmen I

O-Notation erlaubt Kategorisierung der Effizienz von Algorithmen

- $O(1)$: konstante Laufzeit
  - unabhängig von Problemgröße
  - *Beispiel:* Löschen von erstem Element in verketteter Liste
- $O(\log n)$: logarithmische Laufzeit
  - Laufzeit wächst langsamer als Problemgröße
  - typisch für Divide & Conquer Algorithmen (s. nächstes Übungsblatt)
  - *Beispiel:* Suchen in sortierter Liste

Effizienz von Algorithmen I

O-Notation erlaubt Kategorisierung der Effizienz von Algorithmen

- $O(1)$: konstante Laufzeit
  - unabhängig von Problemgröße
  - *Beispiel:* Löschen von erstem Element in verketteter Liste
- $O(\log n)$: logarithmische Laufzeit
  - Laufzeit wächst langsamer als Problemgröße
  - typisch für Divide & Conquer Algorithmen (s. nächstes Übungsblatt)
  - *Beispiel:* Suchen in sortierter Liste
- $O(n)$: lineare Laufzeit
  - Laufzeit wächst vergleichbar zur Problemgröße
  - jedes Eingabe-Element erfordert $O(1)$ Arbeit
  - *Beispiele:* Suchen in unsortierter Liste, Löschen von Element in sequentieller Liste

Effizienz von Algorithmen II

- $O(n \log n)$: "loglinear" Laufzeit
  - Laufzeit wächst schneller als Problemgröße
  - typisch für Divide & Conquer Algorithmen (s. nächstes Übungsblatt)
  - *Beispiele:* Quicksort (s. nächstes Übungsblatt)

Effizienz von Algorithmen II

- $O(n \log n)$: "loglinear" Laufzeit
  - Laufzeit wächst schneller als Problemgröße
  - typisch für Divide & Conquer Algorithmen (s. nächstes Übungsblatt)
  - *Beispiele:* Quicksort (s. nächstes Übungsblatt)
- $O(n^2)$: quadratische Laufzeit
  - typisch für Algorithmen, die Element paarweise kombinieren
  - *Beispiele:* Insertion Sort

Effizienz von Algorithmen II

- $O(n \log n)$: "loglinear" Laufzeit
  - Laufzeit wächst schneller als Problemgröße
  - typisch für Divide & Conquer Algorithmen (s. nächstes Übungsblatt)
  - *Beispiele:* Quicksort (s. nächstes Übungsblatt)
- $O(n^2)$: quadratische Laufzeit
  - typisch für Algorithmen, die Element paarweise kombinieren
  - *Beispiele:* Insertion Sort
- $O(n^3)$: kubische Laufzeit
  - *Beispiel:* Matrix-Matrix Multiplikation

Effizienz von Algorithmen II

- $O(n \log n)$: "loglinear" Laufzeit
  - Laufzeit wächst schneller als Problemgröße
  - typisch für Divide & Conquer Algorithmen (s. nächstes Übungsblatt)
  - *Beispiele:* Quicksort (s. nächstes Übungsblatt)
- $O(n^2)$: quadratische Laufzeit
  - typisch für Algorithmen, die Element paarweise kombinieren
  - *Beispiele:* Insertion Sort
- $O(n^3)$: kubische Laufzeit
  - *Beispiel:* Matrix-Matrix Multiplikation
- $O(2^n)$: exponentielle Laufzeit
  - auch als "unlösbar" bezeichnet (intractable)
  - *Beispiel:* Traveling Salesman (kürzeste Route, so dass alle Städte exakt einmal besucht)

Komplexitätsschätzung **rot13**

	Häufigkeit
rot13(*string*):	
encrypted = *string*;	1
for *i* = 0 **to** **size**(*string*) − 1 {	2*n* + 2
if (*string*[*i*] ≥ 'a' ∧ *string*[*i*] ≤ 'z') {	*n*
abstand = *string*[*i*] − 'a';	t_i
rotierter_abstand = (*abstand* + 13) *mod* 26;	t_i
encrypted[*i*] = 'a' + *rotierter_abstand*;	t_i
}	
}	

mit $n = \textbf{size}(string)$

- Laufzeitfunktion ungefähr

$T(n) = 1 + (2n + 2) + n + t_i + t_i + t_i$, im Worst-Case $t_i = n$
$\Rightarrow O(n)$ lineare Laufzeit

Komplexitätsschätzung **MatrixMultiplication**

	Häufigkeit
MatrixMultiplikation(A,B):	
if(**cols**$(A) \neq$ **rows**(B)){	1
return $-\infty$;	t_c
}	
$n =$ **cols**(A);	1
$C =$ **matrix**(**rows**(A), **cols**(B));	$1 \cdot t_m$
for $i = 0$ **to** **rows**$(A) - 1$ {	$2n + 2$
for $j = 0$ **to** **cols**$(B) - 1$ {	$n \cdot (2n + 2)$
$C[i][j] = 0$;	$n^2 \cdot 1$
for $k = 0$ **to** $n - 1$ {	$n^2 \cdot (2n + 2)$
$C[i][j] = C[i][j] + A[i][k] \cdot B[k][j]$;	$n^3 \cdot 1$
}	
}	
}	

- Laufzeitfunktion ungefähr

$$T(n) = 1 + t_c + 1 + t_m + (2n+2) + 2n^2 + 2n + n^2 \cdot 1 + 2n^3 + 2n^2 + n^3 \cdot 1$$

$\Rightarrow O(n^3)$ kubische Laufzeit

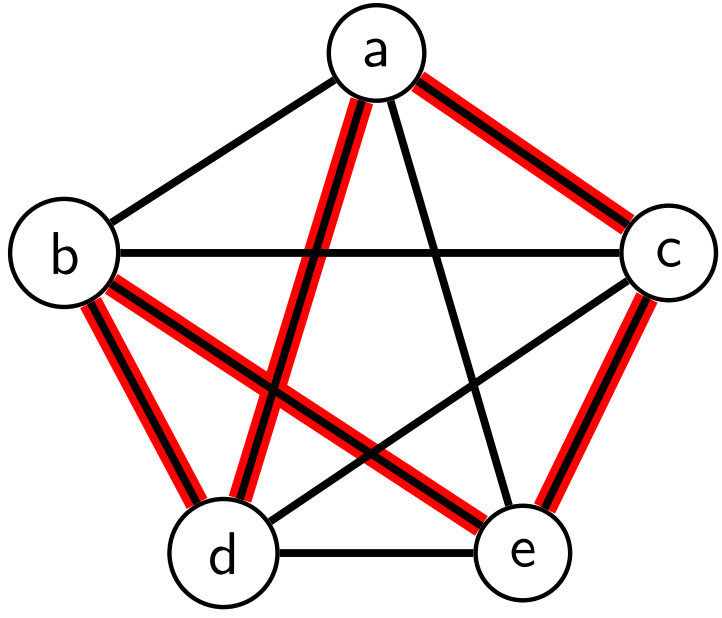
Zusammenfassung

- Komplexitätsabschätzung oft intuitiv
- Beispiel: Eingabelänge n, aber nur lokale Änderung (etwa Anfügung an verkettete Liste) $\Rightarrow O(1)$
- Beispiel: Eingabelänge n, und **for**-Schleife (etwa Minimum in Array) $\Rightarrow O(n)$
- Wichtig ist vor allem Verständnis für die Bedeutung!

- Einfache Algorithmen/Probleme können extrem komplex zu lösen sein!
  - Traveling Salesman Problem
  - Primfaktorzerlegung

Traveling Salesman Problem (TSP)

- Beispiel: Straßenkarte mit Kilometerangaben
- **Problem:** Finde Rundreise, so dass jeder Knoten (jede Stadt) genau einmal besucht wird, und die Gesamtstrecke möglichst kurz ist!



Traveling Salesman Problem (TSP)

- Die Beschreibung des Problems ist trivial!
- Die exakte Lösung aber extrem komplex!
- Verwendet viel Zeit zur expliziten Untersuchung aller Möglichkeiten!