8 Komplexitätstheorie

Formale Grundlagen der Informatik I

Herbstsemester 2012

Robert Marti

Vorlesung teilweise basierend auf Unterlagen
von Prof. emer. Helmut Schauer

Grundidee der Komplexitätstheorie

Gewisse Probleme können gar nicht per Computer gelöst werden
(z.B. das Halteproblem)

Andere Probleme können zwar gelöst werden, brauchen aber unter Umständen zu viel Rechenzeit oder Speicherplatz um in der Praxis exakt gelöst zu werden.

Ausserdem gibt es zur Lösung eines Problems oft verschiedene Methoden (und/oder Datenstrukturen), deren Zeit- oder Platzbedarf sich stark unterscheiden.

$\Rightarrow$ Hilfsmittel zur Untersuchung des (ungefähren) Zeit- und/oder Platzbedarfs erwünscht!

Verschiedene Arten der Analyse

Worst Case Analysis
Ressourcenbedarf im schlechtesten Fall

Average Case Analysis
Ressourcenbedarf im Mittel

Lower Bound Analysis
"besser geht's nicht": untere Schranke für Aufwand

Upper Bound Analysis
"schimmer geht's nimmer" ☺ : obere Schranke für Aufwand

Zeitbedarf für Ausführen eines Programms (Fakultät)

z.B. Zählen "**elementarer Operationen**": Lesen und Schreiben des Werts einer Variable, Addition, Multiplikation, Vergleich, "Sprunganweisung", …

Zeile Code (Fakultät einer Zahl)

```
1    public static long factorial(int n) {
2        long factorial = 1;
3        int i = 1;
4        while ( i <= n ) {
5            factorial = factorial * i;
6            i = i + 1;
7        }
8        return factorial;
9    }
```

Zeitbedarf für Ausführen eines Programms (Fakultät)

z.B. Zählen "**elementarer Operationen**": Lesen und Schreiben des Werts einer Variable, Addition, Multiplikation, Vergleich, "Sprunganweisung", …

Zeile	Code (Fakultät einer Zahl)	#ops	#execs
1	`public static long factorial(int n) {`		
2	`    long factorial = 1;`	1	1
3	`    int i = 1;`	1	1
4	`    while ( i <= n ) {`	4	n
5	`        factorial = factorial * i;`	4	n
6	`        i = i + 1;`	3	n
7	`    }`		
8	`    return factorial;`	1	1
9	`}`		

Totale Anzahl Operationen (in Abhängigkeit der Grösse des Inputs, der Variable n):
$A(n) = 1 + 1 + 4 \cdot n + 4 \cdot n + 3 \cdot n + 1 = 11 \cdot n + 3$ (im wesentlichen also linearer Aufwand)

Bemerkungen zum Beispiel

Im vorherigen Beispiel wurden einige Details unter den Tisch gekehrt …

- Was ist genau der Satz von Operationen des verwendeten Rechners?
- Verschiedene Operationen dauern verschieden lang, z.B.
  - Division > Multiplikation > Addition
  - spezielle Operationen (increment, decrement, shift left + right)
  - double > floating point > long > int

  (> bedeutet "dauert länger")
- Fakultät kann in Java mit Datentyp long nur bis zu einer gewissen Input-Grösse berechnet werden: Resultat < 9.23×10^{18} ⇒ Input ≤ 20

  (In Java könnte die Klasse BigInteger verwendet werden, wobei ... BigInteger > double)

Es geht bei der Aufwandabschätzung jedoch nicht um Details, sondern um das "Big Picture" ... vor allem, wie sich der Aufwand bei der sukzessiven "Vergrösserung" eines Problems verhält.

Zeitbedarf für Ausführen eines Programms (Binäre Suche)

z.B. Zählen von **Funktionsaufrufen**

Zeile Code (binäres Suchen in geordnetem Array)

```
1     public static int search(int a[], int v, l, r) {
2         // assumption: a is in ascending order
3         if ( l > r ) return (-1);  // not found
4         int i = (l + r)/2;
5         if ( v == a[i] ) return i;
6         if ( v < a[i] ) return search(a, v, l, i-1);
7         else return search(a, v, i+1, r);
8     }
```

Aufruf: `int index = search(a, v, 0, n-1);  // length n`

Totale Anzahl Operationen (in Abhängigkeit der Länge des Arrays, n):

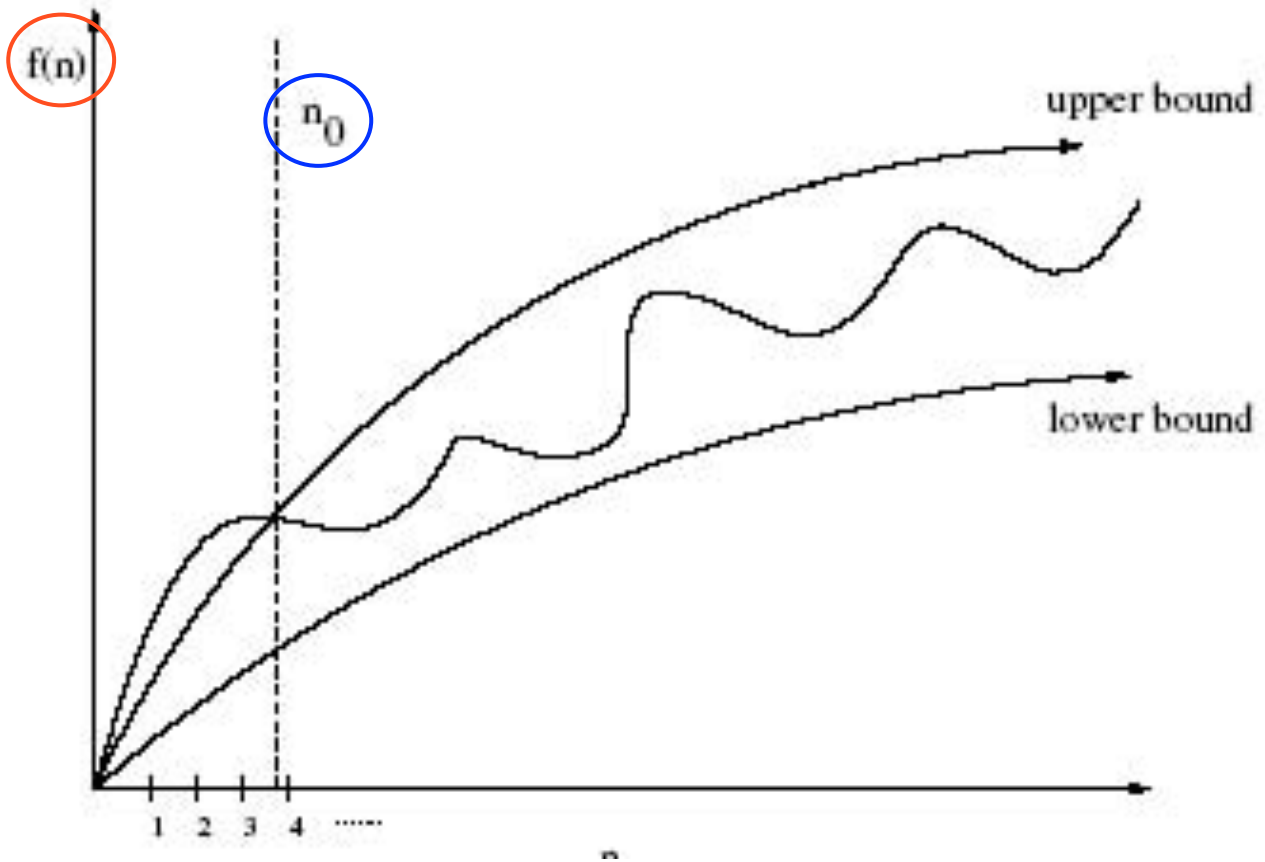
$A(n) = 1 + A(n/2)$ für $n > 0$, $A(1) = 1$ für $n = 0$

$\Rightarrow \quad 2^{A(n)-1} = n \quad \Rightarrow \quad A(n) - 1 = \log_2 n \quad \Rightarrow \quad A(n) = 1 + \log_2 n$

Basis der Aufwandabschätzung

Annahme: Aufwand eines Problems wird als Anzahl von "Operationen" $f(n)$ (z.B. Anzahl Multiplikationen oder Aufrufe einer Prozedur) in Abhängigkeit einer "Eingabegrösse" n (z.B. Grösse einer Zahl / eines Arrays) beschrieben.

Gesucht sind obere (seltener untere) Schranken für $f(n)$ ab einer gewissen "Minimalgrösse" n_0



Ordnung einer Funktion

Ordnung einer Funktion: $\mathrm{O}(f)$
(*order of a function*; auch: "*Big O Notation*")

$$g \in \mathrm{O}(f) \Leftrightarrow \exists\, c, n_0 : c > 0 : (\forall n : n \geq n_0 : g(n) \leq c \cdot f(n))$$

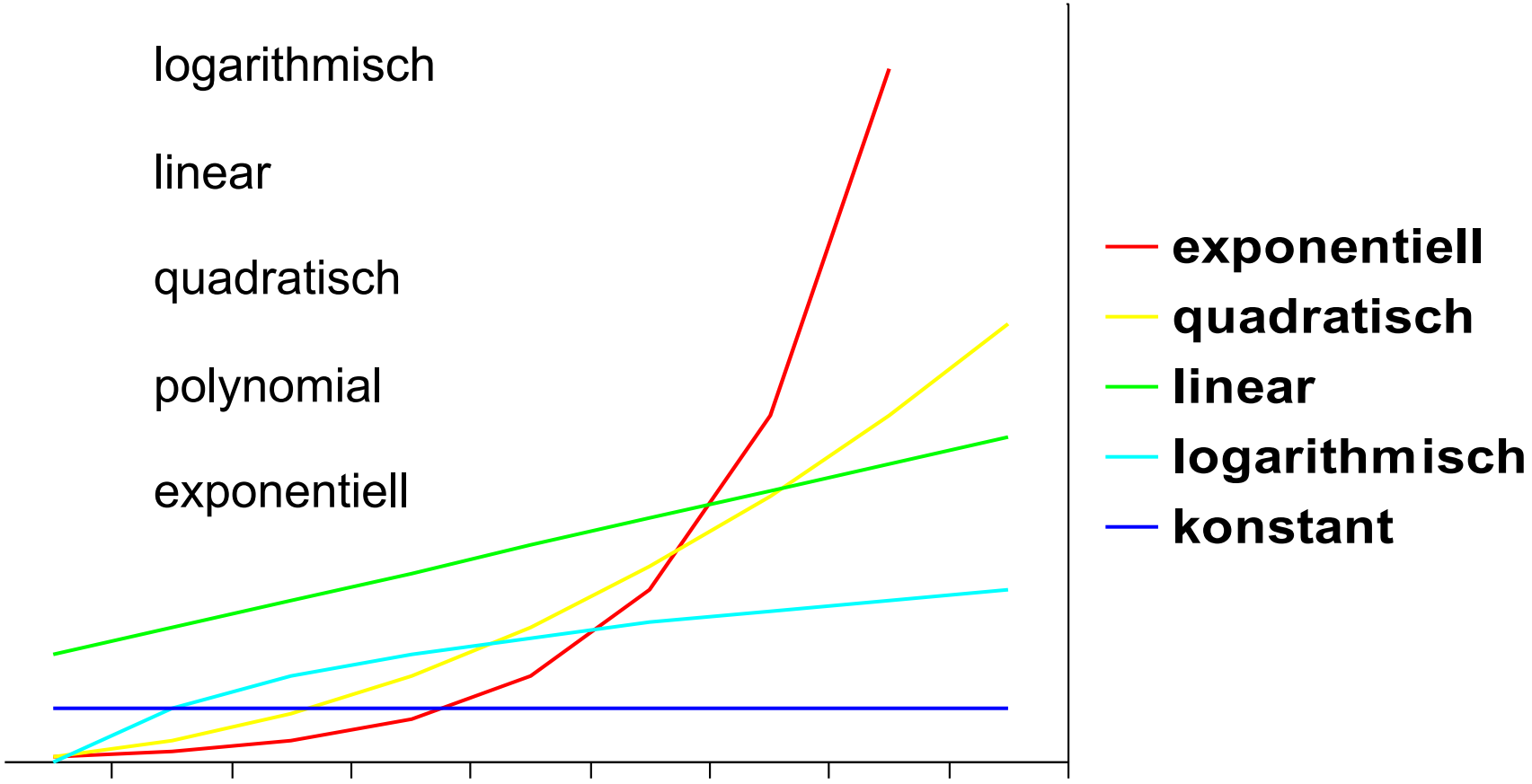
oder

$$g \in \mathrm{O}(f) \Leftrightarrow \lim_{n \to \infty} {}^{g(n)}\!/_{f(n)} = c$$

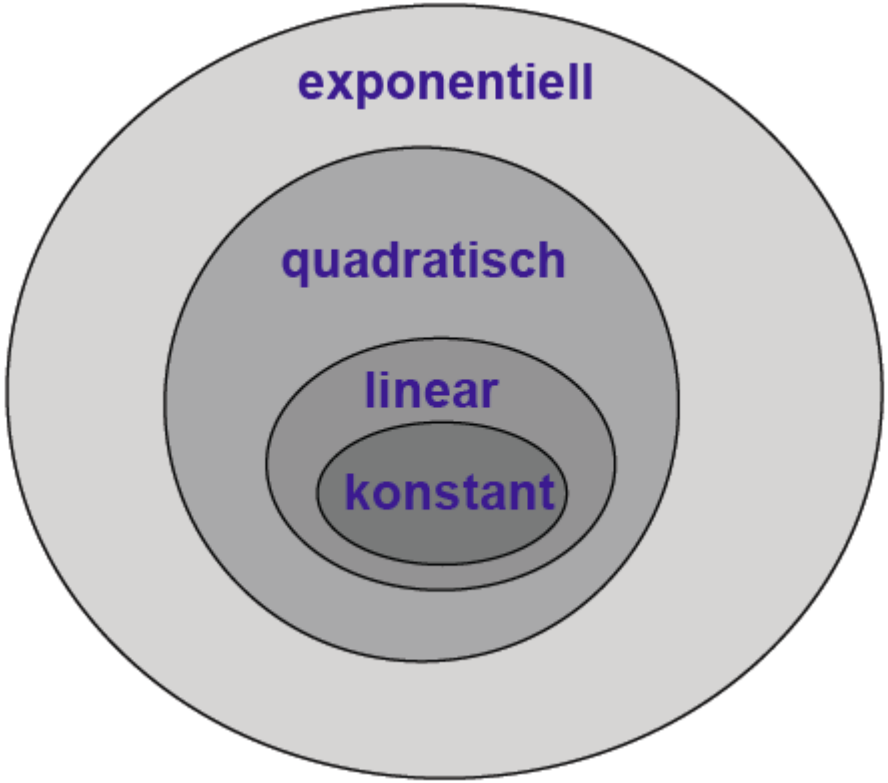
$g \in \mathrm{O}(f)$ ... „g ist von der Ordnung f“

Typische Ordnungen von Funktionen

Notation	Bezeichnung
$O(1)$	konstant
$O(\log n)$	logarithmisch
$O(n)$	linear
$O(n^2)$	quadratisch
$O(n^k)$	polynomial
$O(e^n)$	exponentiell



Mengendarstellung der Ordnung von Funktionen



Mit anderen Worten: Jede Aufwandsfunktion linearer Ordnung ist z.B. auch quadratischer Ordnung …

Natürlich wird versucht, die "**kleinstmögliche**" Ordnung einer Funktion zu ermitteln bzw. anzugeben.

Problemgrössen und Anzahl Operationen

n	ld n	n ld n	n^2	2^n	n!
10	3	33	100	1024	$3*10^6$
20	4	86	400	10^6	$2*10^{18}$
100	7	664	10'000	10^{31}	10^{161}
1000	10	10'000	10^6		
10000	13	130'000	10^8		

zum Vergleich: es gibt etwa 10^{79} Protonen im Universum

Pro Memoria: $\mathrm{ld}\, n = \log_2 n$

Problemgrössen und Zeitaufwand

Annahme: Jede Operation dauert 1 Microsekunde ($1\mu s = 10^{-6s}$)

n	ld n	n ld n	n^2	2^n	n!
10	3 µs	33 µs	100 µs	1 ms	3 s
20	4 µs	86 µs	400 µs	1 s	10^5 Jahre
100	7 µs	664 µs	10 ms	10^{17} Jahre	
1000	10 µs	10 ms	1 s		
10000	13 µs	130 ms	100 s		

zum Vergleich: der Urknall war vor etwa 15 Milliarden Jahren

Rechenregeln für Big-O Notation

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

$$O(f(n) + g(n)) = \max(O(f(n)), O(g(n)))$$

$$O(f(n)) \leq O(g(n)) \Leftrightarrow f(n) \in O(g(n))$$

$$O(f(n)) = O(g(n)) \Leftrightarrow (O(f(n)) \leq O(g(n))) \text{ and } (O(g(n)) \leq O(f(n)))$$

$$O(f(n)) < O(g(n)) \Leftrightarrow (O(f(n)) \leq O(g(n))) \text{ and } (O(g(n)) \neq O(f(n)))$$

Beispiele zu den Rechenregeln

$$O(2 \cdot n - 1) = O(n)$$

$$O(n \cdot {}^{(n+1)}/_{2}) = O(n^2)$$

$$O(\mathrm{ld}\, n) = O(\log n)$$

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

$$O(n \cdot \log n) < O(n^2)$$

$$O(\log n) < O(n^{1/2})$$

P- und NP-Probleme

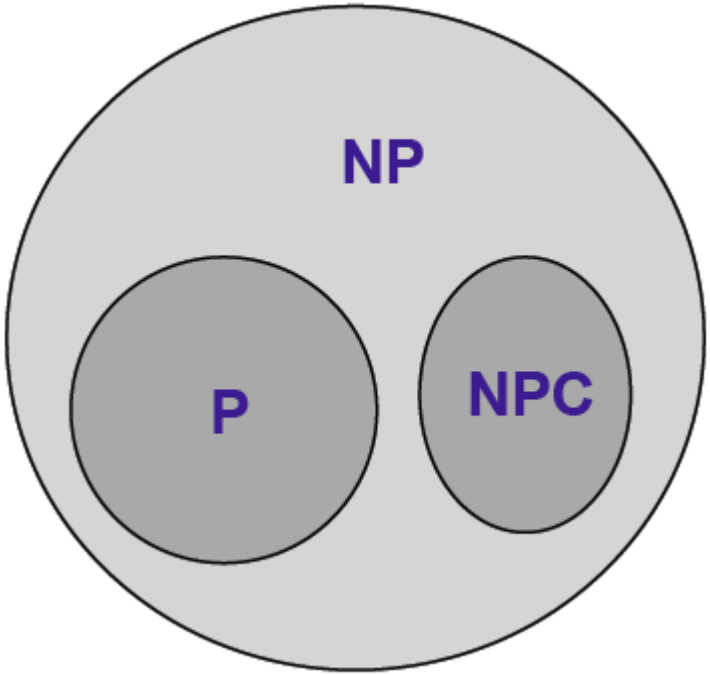
P-Probleme sind mit polynomialem Aufwand lösbar

NP-Probleme sind nicht mit polynomialem Aufwand lösbar

NP steht für “nichtdeterministisch polynomial”

Ob P=NP oder P≠NP ist ist ungelöst!

$P \subseteq NP$



NP-vollständige Probleme

Alle NP-vollständigen Probleme können mit polynomialem Aufwand aufeinander abgebildet werden.

Falls ein einziges der NP-vollständigen Probleme mit polynomialem Aufwand gelöst werden kann gilt P=NP!

Beispiele für NP-vollständige Probleme

Satisfiability

Traveling Salesperson

Knapsack

Scheduling

Bin-Packing

Graph Coloring

Maximal Cliques

Satisfiability

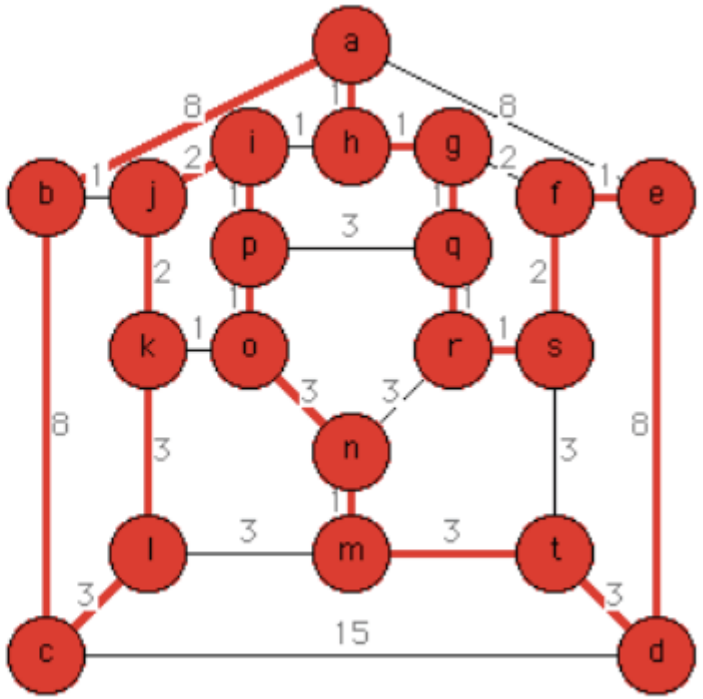
Finde Variablenbelegungen, welche eine Boole'sche Formel erfüllen (wahr machen).

Beispiel: Russian Spy Problem ...

												Line 1				Line 2				Line 3	Line 4	All Lines
RS	GS	SS	RM	GM	SM	RE	GE	SE	RS & GM & GE	GS & RM & GE	GS & GM & RE		RS != GS	RM != GM	RE != GE		!RS | SS	!RM | SM	!RE | SE		RS = GM	
0	0	0	0	0	0	1	0	1	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE
0	0	0	0	0	1	1	0	1	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE
0	0	0	0	1	0	1	0	1	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
0	0	0	0	1	1	1	0	1	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
0	0	0	1	0	0	1	0	1	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
0	0	0	1	0	1	1	0	1	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE
0	0	0	1	1	0	1	0	1	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE
0	0	0	1	1	1	1	0	1	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
0	0	1	0	0	0	1	0	1	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE
0	0	1	0	0	1	1	0	1	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE
0	0	1	0	1	0	1	0	1	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
0	0	1	0	1	1	1	0	1	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
0	0	1	1	0	0	1	0	1	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
0	0	1	1	0	1	1	0	1	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE
0	0	1	1	1	0	1	0	1	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE
0	0	1	1	1	1	1	0	1	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
0	1	0	0	0	0	1	0	1	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE

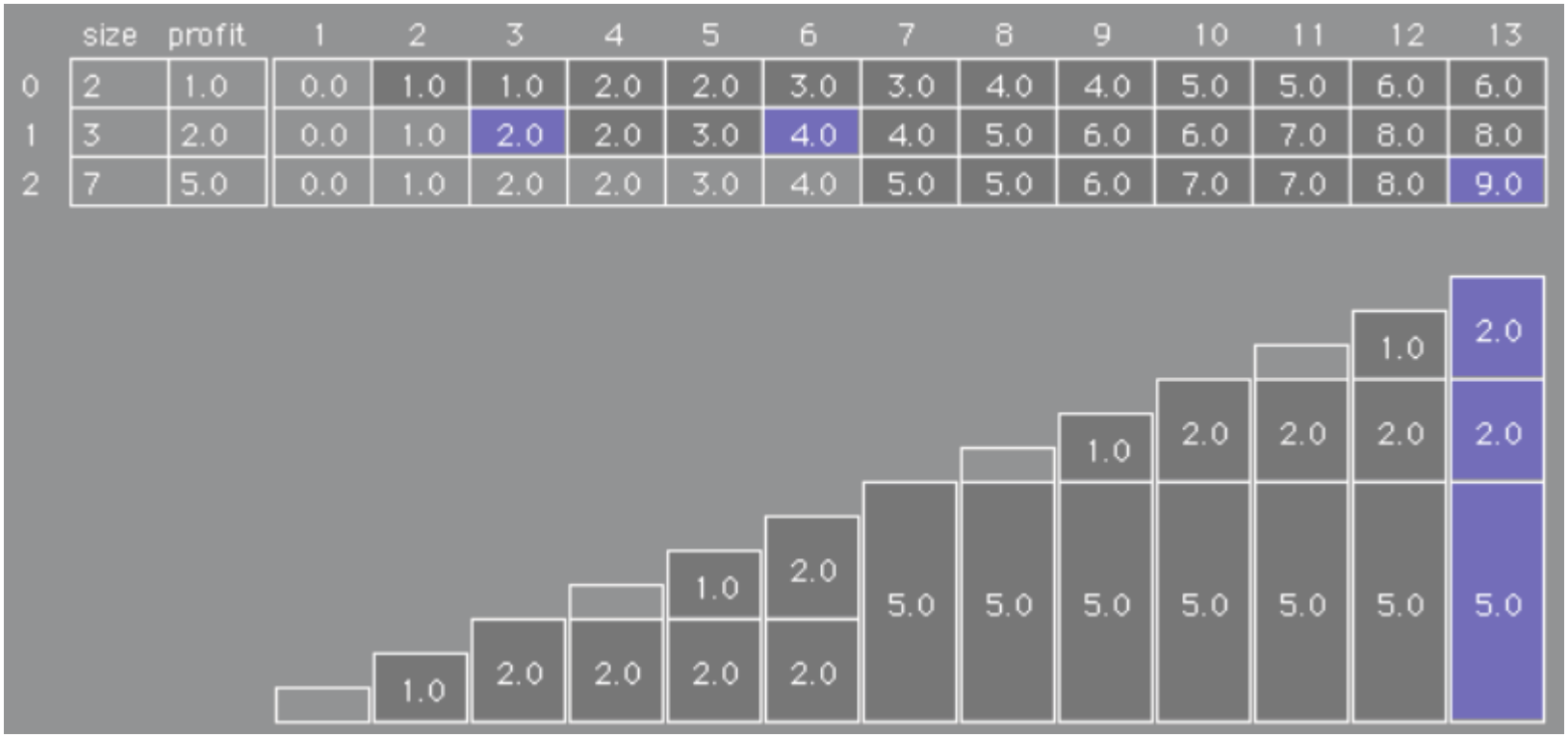
Traveling Salesperson

Finde die kürzeste Route eines Handlungsreisenden (*Traveling Salesperson*), der N Städte besuchen muss, wobei jede Stadt nur einmal besucht werden darf.



Knapsack

Gegeben sind Grösse (*size*) und Wert (*value, profit*) von N Objekten.
Finde die optimale Füllung eines Rucksacks (*knapsack*) gegebener Kapazität mit einer Auswahl dieser Objekte, so dass deren Gesamtwert maximal ist.
Die Objekte dürfen nicht geteilt und die Kapazität des Rucksacks darf nicht überschritten werden.



	size	profit	1	2	3	4	5	6	7	8	9	10	11	12	13
0	2	1.0	0.0	1.0	1.0	2.0	2.0	3.0	3.0	4.0	4.0	5.0	5.0	6.0	6.0
1	3	2.0	0.0	1.0	2.0	2.0	3.0	4.0	4.0	5.0	6.0	6.0	7.0	8.0	8.0
2	7	5.0	0.0	1.0	2.0	2.0	3.0	4.0	5.0	5.0	6.0	7.0	7.0	8.0	9.0

Subset Sum (Teilsummenproblem)

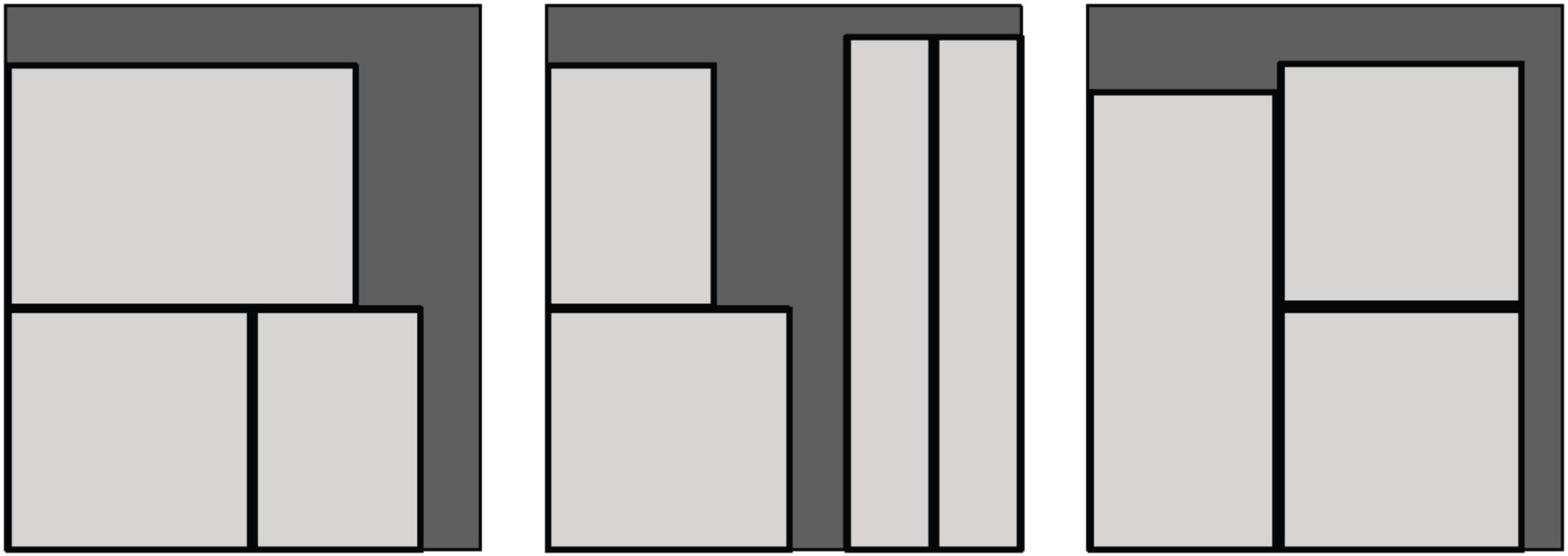
Wähle aus N gegebenen natürlichen Zahlen eine Teilmenge, sodass deren Summe gleich einer ebenfalls gegebenen natürlichen Zahl ist.

Beispiel: Geldwechseln
gegeben: 5,10, 20, 50, 100, 200, 500 Rappenstücke

Bin Packing

Packe N gegebene Schachteln (bins) so in Container gegebener Grösse, dass eine minimale Anzahl von Containern benötigt wird.

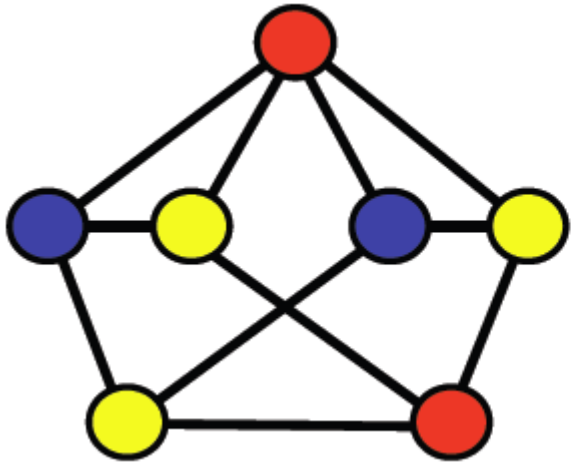
Alternative:
Ausschneiden verschiedner Formen aus vorgegebenen Stücken

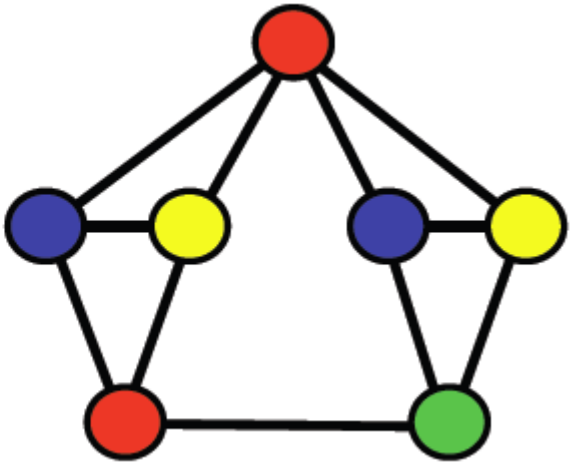


Graph Coloring

Können die N Knoten eines gegebenen Graphen so mit drei Farben gefärbt werden, dass benachbarte Knoten unterschiedlich gefärbt sind?

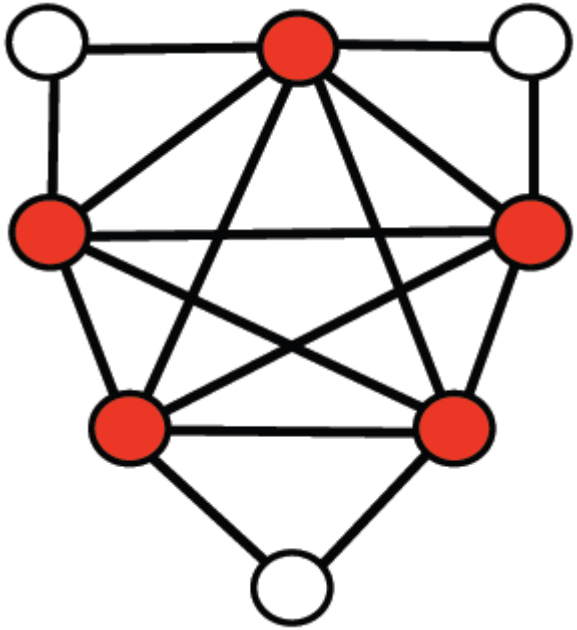
vgl. auch Einfärben von Karten





Maximal Cliques

Finde in einem gegebenen Graphen jene maximale Teilmenge von Knoten, von denen jeder Knoten mit allen anderen Knoten dieser Teilmenge verbunden ist.
(Eine Clique ist eine Menge von Personen, von denen jede Person alle anderen Personen der Clique kennt.)



Nicht berechenbare Probleme

Es gibt Aufgaben die nicht algorithmisch lösbar sind.

Beispiel: **Halteproblem**

Es gibt keinen Algorithmus der entscheidet ob ein beliebiges Programm terminiert!

Indirekter Beweis zum Halteproblem

Annahme:
Es gibt eine Boole'sche Funktionsprozedur T(P), die genau dann den Wert true liefert, wenn ihr Argument, eine beliebige Prozedur, P terminiert.

Die Prozedur P kann zum Beispiel auch so definiert werden:

procedure P { while T(P) {} }

Widerspruch:
Falls P terminiert

- dann ergibt T(P) den Wert true
- aber dann kann P nicht terminieren (endlose while-Schleife)!

Daraus folgt: Die obige Annahme ist unmöglich!
Es gibt somit keine solche Funktionsprozedur T(P).