1. (3 Punkte) Ein Rechner schafft 10^9 Schritte pro Sekunde.
   (a) Wie lange brauchen Algorithmen mit n^2, n^3 und 2^n Schritten für $n = 20$ und für $n = 40$?
   (b) Um welchen Faktor wächst die Laufzeit jeweils, wenn sich n verdoppelt?

Lösung:

(a) $n = 20$: n^2: $4 \cdot 10^{-7}$ s, n^3: $8 \cdot 10^{-6}$ s, 2^n: 1048576 Schritte, etwa 0.001 s.
$n = 40$: n^2: $1.6 \cdot 10^{-6}$ s, n^3: $6.4 \cdot 10^{-5}$ s, $2^n \approx 1.1 \cdot 10^{12}$ Schritte, etwa 1100 s, also gut 18 Minuten.

(b) n^2: Faktor 4. n^3: Faktor 8. 2^n: Faktor $2^{20} \approx 10^6$, die Laufzeit wird quadriert.

2. (3 Punkte) Formuliere die folgenden Optimierungsprobleme als Entscheidungsprobleme.
   (a) Finde die kürzeste Rundreise durch 20 Städte.
   (b) Wie viele Farben braucht man mindestens, um eine Landkarte zu färben?
   (c) Packe Gegenstände mit möglichst großem Wert in einen Rucksack mit begrenzter Tragfähigkeit.

Wie kann man mit einem schnellen Algorithmus für das Entscheidungsproblem in (a) die Länge der kürzesten Rundreise bestimmen?

Lösung:

(a) Gibt es eine Rundreise, die kürzer als k ist?

(b) Lässt sich die Karte mit k Farben färben?

(c) Gibt es eine Auswahl der Gegenstände, die höchstens die Tragfähigkeit wiegt und mindestens den Wert k hat?

Man stellt die Frage für verschiedene k und sucht mit binärer Suche das kleinste k, für das die Antwort Ja ist. Das braucht nur logarithmisch viele Fragen.

3. (4 Punkte) Gib jeweils an, in welcher der Klassen P, NP, R das Problem liegt, soweit man es weiß, und begründe.
   (a) Gibt es in einem Graphen einen Weg von u nach v?
   (b) Hat dieses Sudoku eine Lösung? (verallgemeinert auf beliebig große Felder)
   (c) Hat die Zahl n einen Teiler zwischen 2 und k?
   (d) Hält dieses Programm mit dieser Eingabe an?

Lösung:

(a) In P (und damit in NP und R): Breitensuche findet den Weg in polynomieller Zeit.

(b) In NP: Eine vorgeschlagene Lösung prüft man schnell (Zeilen, Spalten, Blöcke). Das Problem ist sogar NP-vollständig, ein polynomieller Algorithmus ist nicht bekannt.

(c) In NP: Ein vorgeschlagener Teiler wird mit einer Division geprüft. Ob das Problem in P liegt, weiß man nicht.

(d) Nicht in R: Das Halteproblem ist unentscheidbar, also auch nicht in NP und nicht in P.

4. (4 Punkte) Gegeben ist die Formel

$$(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee x_4) \wedge (x_1 \vee \overline{x_2})$$

   (a) Benenne die Klauseln und Literale.
   (b) Werte die Formel für $x_1 = True$, $x_2 = False$, $x_3 = True$, $x_4 = False$ aus und für $x_1 = True$, $x_2 = True$, $x_3 = False$, $x_4 = True$.
   (c) Wie viele Belegungen muss ein Brute-Force-Algorithmus höchstens prüfen? Wie viele sind es bei 30 Variablen?
   (d) Warum ist SAT in NP?

Lösung:

(a) Drei Klauseln in Klammern, verbunden mit UND. Literale sind z.B. $\overline{x_1}$, x_2, x_3 in der ersten Klausel.

(b) Erste Belegung: True, False, True für die drei Klauseln, die Formel ist False. Zweite Belegung: True, True, True, die Formel ist True.

(c) $2^4 = 16$ Belegungen (davon sind 8 erfüllend). Bei 30 Variablen $2^{30} \approx 10^9$.

(d) Eine vorgeschlagene Belegung lässt sich schnell prüfen: Man wertet jede Klausel einmal aus, die Zeit wächst linear mit der Länge der Formel.

5. (3 Punkte) Zeige, dass die Formel

$$(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_2})$$

nicht erfüllbar ist. Warum ist es für ein Entscheidungsverfahren schwieriger, die Antwort Nein zu begründen als die Antwort Ja?

Lösung: Jede der vier Belegungen von x_1, x_2 macht genau eine Klausel falsch: $x_1 = x_2 =$ False die erste, $x_1 =$ True, $x_2 =$ False die zweite, $x_1 =$ False, $x_2 =$ True die dritte, $x_1 = x_2 =$ True die vierte.

Für ein Ja genügt eine erfüllende Belegung als Nachweis, sie ist schnell geprüft. Für ein Nein kennt man im Allgemeinen keinen kurzen Nachweis, man muss im schlimmsten Fall alle 2^n Belegungen ausschließen.

6. (4 Punkte) Der Graph hat die Knoten A, B, C und die Kanten A–B und B–C. Die Frage, ob er sich mit 2 Farben färben lässt, soll in eine SAT-Formel übersetzt werden. Die Variable $x_{v,c}$ bedeutet: Knoten v hat Farbe c ($c = 1, 2$).
   (a) Schreibe die Klauseln auf: Jeder Knoten hat mindestens eine Farbe, kein Knoten hat zwei Farben, benachbarte Knoten haben nicht dieselbe Farbe. Wie viele Variablen und Klauseln sind es?
   (b) Gib eine erfüllende Belegung an und die zugehörige Färbung.
   (c) Warum muss die Übersetzung selbst schnell (in polynomieller Zeit) gehen?

Lösung:

(a) $(x_{A,1} \vee x_{A,2})$, $(x_{B,1} \vee x_{B,2})$, $(x_{C,1} \vee x_{C,2})$, $(\overline{x_{A,1}} \vee \overline{x_{A,2}})$, $(\overline{x_{B,1}} \vee \overline{x_{B,2}})$, $(\overline{x_{C,1}} \vee \overline{x_{C,2}})$, $(\overline{x_{A,1}} \vee \overline{x_{B,1}})$, $(\overline{x_{A,2}} \vee \overline{x_{B,2}})$, $(\overline{x_{B,1}} \vee \overline{x_{C,1}})$, $(\overline{x_{B,2}} \vee \overline{x_{C,2}})$.
6 Variablen, 10 Klauseln.

(b) $x_{A,1} = x_{B,2} = x_{C,1} =$ True, alle anderen False: A und C haben Farbe 1, B hat Farbe 2.

(c) Sonst könnte die Übersetzung selbst die schwere Arbeit machen. Nur wenn sie schnell ist, folgt aus einem schnellen SAT-Algorithmus auch ein schneller Algorithmus für das Färben.

7. (3 Punkte) Nach dem kleinen Satz von Fermat gilt für jede Primzahl p und $1 \leq a < p$: $a^{p-1} \,\%\, p = 1$.
   (a) Berechne $2^8 \,\%\, 9$ und $2^{10} \,\%\, 11$. Was folgt daraus für 9 und 11?
   (b) Es gilt $2^{560} \,\%\, 561 = 1$, aber $561 = 3 \cdot 11 \cdot 17$. Was bedeutet das für einen Primzahltest, der nur $2^{n-1} \,\%\, n$ berechnet?
   (c) Warum ist der Fermat-Test ein Beispiel für „Suchen ohne zu suchen“?

Lösung:

(a) $2^8 = 256 = 28 \cdot 9 + 4$, also $2^8 \,\%\, 9 = 4$. 9 ist keine Primzahl. $2^{10} = 1024 = 93 \cdot 11 + 1$, also $2^{10} \,\%\, 11 = 1$. Das passt zu einer Primzahl, beweist es aber nicht.

(b) Ein solcher Test hält 561 fälschlich für eine Primzahl. Das Ergebnis 1 ist kein Beweis. Erst der AKS-Test (2002) entscheidet in polynomieller Zeit sicher.

(c) Ist das Ergebnis nicht 1, weiß man, dass die Zahl keine Primzahl ist, ohne einen Teiler gefunden zu haben.

8. (4 Punkte) (a) Was bedeutet es, dass ein Problem NP-vollständig ist?
   (b) Was hat Stephen Cook 1971 bewiesen und warum ist das wichtig?
   (c) Was würde folgen, wenn jemand einen polynomiellen Algorithmus für SAT findet?
   (d) Warum benutzt man für NP-vollständige Probleme in der Praxis oft Näherungsverfahren?

Lösung:

(a) Es liegt in NP, und jedes Problem in NP lässt sich in polynomieller Zeit darauf reduzieren (es ist NP-schwer). Es gehört also zu den schwersten Problemen in NP.

(b) SAT ist NP-vollständig. Damit gab es ein erstes NP-vollständiges Problem. Für weitere Probleme genügt es seitdem zu zeigen, dass sie in NP liegen und SAT sich auf sie reduzieren lässt.

(c) Dann ließe sich jedes Problem in NP in polynomieller Zeit lösen: erst auf SAT reduzieren, dann SAT lösen. Es wäre P = NP.

(d) Man kennt nur exponentielle Algorithmen, die schon bei mittelgroßen Eingaben zu lange brauchen. Ein Näherungsverfahren wie Greedy liefert in polynomieller Zeit eine Lösung, die oft gut, aber nicht immer optimal ist.

9. (4 Punkte) Der Graph hat die Knoten a1, a2, a3, b1, b2, b3. Jedes a_i ist mit jedem b_j verbunden, außer wenn $i = j$ ist. Greedy färbt die Knoten in einer Reihenfolge und gibt jedem Knoten die kleinste Farbnummer (0, 1, 2, ...), die keiner seiner schon gefärbten Nachbarn hat.
   (a) Färbe in der Reihenfolge a1, b1, a2, b2, a3, b3. Wie viele Farben braucht Greedy?
   (b) Färbe in der Reihenfolge a1, a2, a3, b1, b2, b3. Wie viele Farben braucht Greedy jetzt?
   (c) Was folgt daraus für Greedy als Verfahren?

Lösung:
   (a) a1: 0, b1: 0, a2: 1, b2: 1, a3: 2, b3: 2, also 3 Farben.
   (b) a1: 0, a2: 0, a3: 0, b1: 1, b2: 1, b3: 1, also 2 Farben.
   (c) Das Ergebnis hängt von der Reihenfolge ab. Greedy ist schnell, findet aber nicht immer die kleinste Zahl von Farben. Es ist ein Näherungsverfahren.

10. (3 Punkte) Ein Graph hat einen Mittelpunkt M mit vier Nachbarn A, B, C, D. Jeder davon hat noch einen weiteren Nachbarn: A–E, B–F, C–G, D–H. Gesucht ist eine möglichst kleine dominierende Menge: Jeder Knoten soll in der Menge liegen oder einen Nachbarn in der Menge haben.
   (a) Wende Greedy an: Wähle immer den Knoten, der die meisten noch nicht abgedeckten Knoten abdeckt (sich selbst und seine Nachbarn). Bei Gleichstand den alphabetisch ersten.
   (b) Gib eine kleinere dominierende Menge an.

Lösung:
   (a) Zuerst M (deckt 5 Knoten ab), danach bleiben E, F, G, H offen, jeder braucht einen eigenen Knoten. Greedy: M, A, B, C, D, also 5 Knoten.
   (b) A, B, C, D: 4 Knoten decken alles ab, auch M.

11. (4 Punkte) Schreibe eine Funktion `greedy_nach_grad(G)`, die einen Graphen (Dictionary mit Nachbarlisten) mit Greedy färbt und die Knoten dabei nach der Zahl ihrer Nachbarn absteigend sortiert, bei Gleichstand alphabetisch. Sie soll die Färbung zurückgeben. Teste sie mit dem Graphen aus der Aufgabe zur Greedy-Färbung.

Lösung:

```
def greedy_nach_grad(G):
    reihenfolge = sorted(G, key=lambda v: (-len(G[v]), v))
    farbe = {}
    for v in reihenfolge:
        belegt = {farbe[w] for w in G[v] if w in farbe}
        c = 0
        while c in belegt:
            c += 1
        farbe[v] = c
    return farbe

krone = {'a1': ['b2', 'b3'], 'a2': ['b1', 'b3'], 'a3': ['b1', 'b2'],
         'b1': ['a2', 'a3'], 'b2': ['a1', 'a3'], 'b3': ['a1', 'a2']}
f = greedy_nach_grad(krone)
print(f, max(f.values()) + 1)
```

Alle Knoten haben zwei Nachbarn, die Reihenfolge ist dann alphabetisch: a1, a2, a3, b1, b2, b3. Greedy braucht so 2 Farben. Die Regel nach Grad hilft oft, garantiert aber auch keine optimale Färbung.