universität wien

Vom Problem zum Programm

Um für eine bestimmte Problemstellung einen Algorithmus zu finden, muss man prinzipiell selbst in der Lage sein, das Problem zu lösen.

Oft ist es hilfreich, kleine Probleminstanzen "mit Papier und Bleistift" zu lösen, die zur Lösung verwendeten Schritte zu beobachten und dann in einem Programm zu formulieren.

universität wien

Beispiel: Primzahlen erkennen (1)

Aufgabe: Schreiben Sie ein Programm, das eine ganze Zahl einliest und ausgibt, ob die eingelesene Zahl eine Primzahl ist oder nicht.

Schritt 1

Finden Sie die Primzahlen aus folgender Liste:

594, 323, 461, 437, 387, 223, 813, 221, 310, 397, 270

universität wien

Beispiel: Primzahlen erkennen (2)

Schritt 2

Ihre Lösung ist: 223, 397, 461?

Wie haben Sie das herausgefunden?

universität wien

Beispiel: Primzahlen erkennen (2)

Schritt 2

Ihre Lösung ist: 223, 397, 461?

Wie haben Sie das herausgefunden?

Wahrscheinliche Vorgehensweise:

Primzahlzerlegung (Testen auf Teilbarkeit mit immer größeren Primzahlen)

Beispiel: Primzahlen erkennen (3)

Schritt 3

Umsetzung im Programm:

Problem 1: Computer kennt keine Primzahlen

Wir testen einfach alle möglichen Zahlen: 2, 3, 4, ...

Computer ist schnell genug, um den erhöhten Aufwand zu bewältigen

Wo können wir aufhören zu testen?

Problem 2: Wie erkennt der Computer, ob eine Zahl ein Teiler ist?

Beispiel: Primzahlen erkennen (3)

Schritt 3

Umsetzung im Programm:

Problem 1: Computer kennt keine Primzahlen

Wir testen einfach alle möglichen Zahlen: 2, 3, 4, ...

Computer ist schnell genug, um den erhöhten Aufwand zu bewältigen

Wo können wir aufhören zu testen?

Problem 2: Wie erkennt der Computer, ob eine Zahl ein Teiler ist?

Umformulierung:

Eine Zahl x teilt y genau dann, wenn die Division von y durch x Rest 0 ergibt!

Das lässt sich direkt in C++ formulieren!

universität wien

Beispiel: Primzahlen erkennen (4)

Schreiben Sie das Programm

Beispiel: Primzahlen erkennen (5)

Eine mögliche Lösung:

```
#include<cmath>
#include<iostream>
using namespace std;
int main()
{
  int n;
  cout << "Geben Sie bitte eine Zahl ein: ";
  cin >> n;

  int i {2};
  while (i <= sqrt(n)) { //wenn n=a*b, muss mindestens einer der Faktoren <= sqrt(n) sein
    if (n%i == 0) {
      cout << n << " ist keine Primzahl\n";
      return 0;
    }
    ++i;
  }

  if (n <= 1) cout << n << " ist keine Primzahl\n"; //Sonderfall 1 und Fehleingaben ausschließen
  else cout << n << " ist eine Primzahl\n";
  return 0;
}
```

Anmerkung: sqrt liefert double Werte und ist damit inhärent ungenau. Eventuell werden im Extremfall Werte falsch als Primzahlen identifiziert (Welche Werte sind das?).
Zur Sicherheit sollten wir also zur Quadratwurzel noch eins addieren.

Programmeffizienz

Unser Programm ist nicht besonders effizient

Allgemein gilt: Optimierung erst, wenn das Programm funktioniert

Eine sehr simple Optimierung wäre hier z.B.:

nur 2 und die ungeraden Zahlen als mögliche Teiler testen

(also i=i+2 statt ++i und 2 als Sonderfall vor die Schleife ziehen)

Berechnung von sqrt vor die Schleife ziehen

Das sind aber nur eine marginale Verbesserung (da die Laufzeit exponentiell steigt)

Wenn man alle Primzahlen in einem Bereich benötigt, sind Siebverfahren vorteilhaft

Sieb des Eratosthenes

Effizientere Primzahltests

Probabilistisch

Fermatscher Primzahltest, Solovay-Strassen Test, Miller-Rabin Test, ...

Für spezielle Arten von Primzahlen

Lucas-Test, Lucas-Lehmer Test, ...

AKS-Methode (2002)

Deterministisch, für extrem große Primzahlen effizienter, eher nur mathematisch interessant (polynomiale Laufzeit)

Große Primzahlen für Verschlüsselungstechniken wichtig!

https://de.wikipedia.org/wiki/Primzahltest

universität wien

Beispiel: Minimumsuche (1)

Suchen Sie das Minimum aus der folgenden Liste von Zahlen:

839 719 728 904 648 419 277 175 181 798 145 537 460 577 864
840 126 518 845 191 459 376 325 389 809 769 992 715 854 563
841 872 813 815 729 189 230 870 478 798 905 806

universität wien

Beispiel: Minimumsuche (2)

Ihre Lösung ist: 126?

Wie haben Sie diese gefunden?

Beispiel: Minimumsuche (3)

Schreiben Sie ein Programm, das beliebig viele ganzzahlige Werte einliest und am Ende den kleinsten eingelesenen Wert ausgibt.

Verwenden Sie einen anderen Programmnamen als zuvor

Vermeiden Sie min oder max als Namen in Ihrem Programm (sind in std schon definiert)

Verwenden Sie zur Eingabe eine Schleife der Form:

```
while (cin >> wert) {/*...*/}
```

Die Eingabeoperation liefert als Ergebnis `cin`

`cin` als logischer Wert interpretiert gibt true, so lange kein Eingabefehler passiert und die Eingabe nicht zu Ende ist (EOF, End of File wird gelesen)

EOF in der Eingabe kann unter Linux durch Drücken von Ctrl-D (Windows Ctrl-Z) erzeugt werden.

Beispiel: Minimumsuche (4)

Mögliche Lösung:

```
#include<iostream>
using namespace std;
int main()
{
  int minimum;
  cin >> minimum; //Annahme: Es wird mindestens ein gueltiger Wert eingegeben

  int n;
  while (cin >> n)
    if (n < minimum)
      minimum = n;

  cout << "Das Minimum ist " << minimum << '\n';
  return 0;
}
```

Programmeffizienz

Unser Programm ist effizient

Es gibt keine Möglichkeit, das Minimum zu finden, ohne nicht jeden Wert mindestens einmal anzusehen.

Sortieren wäre ineffizienter!

Was, wenn das zweitkleinste, drittkleinste, Element gesucht wird?

Bei zwei, drei, etc. könnte man mehrmals Minimumsuche anwenden

Bei n/2-kleinste, n-kleinste, etc. wäre das wieder ineffizient

quickselect, introselect

universität wien

Test mit 4000 Zahlen

Die Datei /home/Xchange/ue2/input_pr1 enthält 4000 Zufallszahlen

Wir können die Eingabe von der Tastatur auf diese Datei umlenken

./minimum </home/Xchange/ue2/input_pr1

Das liefert die kleinste Zahl in der Datei, unter der Annahme, dass das Programm minimum heißt.

universität wien

Beispiel: Suche die maximale Primzahl

Problem in Teile zerlegen, die einzeln lösbar, bzw. deren Lösungen schon bekannt sind (divide and conquer)

Wir wissen schon, wie wir eine Primzahl erkennen

Wir wissen schon, wie wir das Minimum einer Menge von Zahlen finden

Analoge Problemstellungen und deren eventuell schon bekannten Lösungen untersuchen

Maximumsuche ist im Prinzip dasselbe wie Minimumsuche

Programm muss nur leicht adaptiert werden.

Damit unsere Programmteile zusammenarbeiten können, wandeln wir den Primzahltest in eine Funktion um

Primzahltest als Funktion (1)

```
bool ist_prim(int n)
{
/* Hier kommt unser adaptierter Primzahltest her */
}
```

Funktionsdefinition

Adaptionen:

- Keine Eingabe (der zu prüfende Wert kommt als Parameter)
- Keine Ausgabe (das Ergebnis wird retourniert)
- Anpassen der return Statements, sodass true oder false geliefert wird

Primzahltest als Funktion (2)

```
#include<cmath>
#include<iostream>
using namespace std;
int main()
{
  int n;
  cout << "Geben Sie bitte eine Zahl ein: ";
  cin >> n;
  int i {2};
  int stop {static_cast<int>(sqrt(n))};
  while (i <= stop) { //wenn n=a*b, muss mindestens einer der Faktoren <= sqrt(n) sein
    if (n%i == 0) {
      cout << n << " ist keine Primzahl\n";
      return 0;
    }
    ++i;
  }
  if (n <= 1) cout << n << " ist keine Primzahl\n"; //Sonderfall 1 und Fehleingaben ausschließen
  else cout << n << " ist eine Primzahl\n";
  return 0;
}
```

nicht mehr nötig

return false

return n>1
bitte nicht:
```
if (n>1) return true;
else return false;
```

Primzahltest als Funktion (3)

```
bool ist_prim(int n)
{
  int i {2};
  int stop {static_cast<int>(sqrt(n))};
  while (i <= stop) { //wenn n=a*b, muss mindestens einer der Faktoren <= sqrt(n) sein
    if (n%i == 0)
      return false;
    ++i;
  }

  return n>1;
}
```

Anmerkungen:

Da sqrt verwendet wird, muss im Programm weiterhin cmath inkludiert werden.

Bevor die Funktion verwendet werden kann, muss sie zumindest deklariert werden.

Wir fügen die Funktionsdefinition (jede Definition ist auch eine Deklaration) daher vor main ein.

Maximumsuche mit Primzahltest (1)

```
#include<cmath>
#include<iostream>
using namespace std;

bool ist_prim(int n)
{
  int i {2};
  int stop {static_cast<int>(sqrt(n))};
  while (i <= stop) { //wenn n=a*b, muss mindestens einer der Faktoren <= sqrt(n) sein
    if (n%i == 0)
      return false;
    ++i;
  }

  return n>1;
}

int main()
{
  int maximum;
  cin >> maximum; //Annahme: Es wird mindestens ein gueltiger Wert eingegeben

  int n;
  while (cin >> n)
    if (n > maximum)
      maximum = n;

  cout << "Die maximale Primzahl ist " << maximum << '\n';
  return 0;
}
```

universität wien

Maximumsuche mit Primzahltest (2)

Verbinden der beiden Teile

Berücksichtige für Maximumsuche nur Werte für die is_prim true liefert

Problem:

Der erste Wert muss keine Primzahl sein

Wir brauchen also zum Finden des Startwerts auch eine Schleife

Eventuell ist überhaupt keine der eingelesenen Zahlen eine Primzahl

Lösungsansatz:

Eine boolesche Variable zeigt an, ob bereits eine Primzahl gefunden wurde.

Diese wird gegebenenfalls in der ersten Schleife auf true gesetzt.

Falls am Ende die Variable noch immer false ist, gibt es kein Maximum

Maximumsuche mit Primzahltest (3)

```
int main()
{
  bool prim_gefunden {false};
  int maximum;
  int n;
  while (cin >> n)
```

falls n eine Primzahl ist, setze prim_gefunden auf true, maximum auf n und beende die Schleife

```
  while (cin >> n)  //weiterlesen (falls nicht schon EOF)
```

falls n eine Primzahl ist und größer als das bisherige Maximum

```
    maximum = n;
```

falls Primzahl gefunden wurde, gib das Maximum aus, ansonst eine entsprechende Meldung

```
  return 0;
}
```

Pseudocode

Ausbrechen aus einer Schleife mit `break;`

Mögliche Lösung

```
#include<cmath>
#include<iostream>
using namespace std;

bool ist_prim(int n)
{
  int i {2};
  int stop {static_cast<int>(sqrt(n))};
  while (i <= stop) { //wenn n=a*b, muss mindestens einer der Faktoren <= sqrt(n) sein
    if (n%i == 0)
      return false;
    ++i;
  }

  return n>1;
}

int main()
{
  bool prim_gefunden {false};
  int maximum; //Initialisierung nicht noetig (im Allgemeinen auch nicht moeglich)

  int n;
  while (cin >> n)
    if (ist_prim(n)) {
      maximum = n;
      prim_gefunden = true; //gleichbedeutend mit maximum ist initialisiert
      break;
    }

  while (cin >> n) //Schleife wird nur ausgefuehrt, wenn prim_gefunden == true
    if (ist_prim(n) && n > maximum)
      maximum = n;

  if (prim_gefunden)
    cout << "Die maximale Primzahl ist " << maximum << '\n';
  else
    cout << "Es wurde keine Primzahl gefunden!\n";
  return 0;
}
```

universität wien

Muster erzeugen

Erstellen Sie ein Programm, das die beiden Ausgabestatements `cout<<"*";` und `cout<<"\n";` nur jeweils einmal verwendet und folgendes Muster ausgibt:

```
*****
****
***
**
*
```

Problemreduktion: Finde durch Weglassen oder Hinzufügen von Einschränkungen (constraints) eine einfacher lösbare Version des Problems. Vielleicht lässt sich die Lösung des reduzierten Problems auf das ursprüngliche Problem anpassen.

Problemreduktion (1)

Wenn wir beliebige Outputstatements zulassen, wird die Lösung trivial

Wenn wir die beiden Statements beliebig oft zulassen, wird die Lösung lang, aber immer noch trivial

Beide Reduktionen bringen uns mit dem eigentlichen Problem nicht weiter

Problemreduktion (2)

Versuchen wir ein verändertes Muster

*****

*****

*****

*****

*****

Dieses Problem kann man weiter zerlegen:

Gib eine Zeile mit fünf Sternen aus.

Mache das fünf mal.

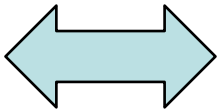
Das sieht machbar aus.

Schreiben Sie das Programm

Problemreduktion (3)

Mögliche Lösung

```
#include<iostream>
using namespace std;
int main()
{
  int zeile {0};
  while (zeile < 5) {
    int spalte {0};
    while (spalte < 5) {
      cout << '*';
      ++spalte;
    }
    cout << '\n';
    ++zeile;
  }
  return 0;
}
```



```
#include<iostream>
using namespace std;
int main()
{
  for (int zeile {0}; zeile < 5; ++zeile) {
    for (int spalte {0}; spalte < 5; ++spalte)
      cout << '*';
    cout << '\n';
  }
  return 0;
}
```

Problemreduktion (4)

Betrachten wir nun die Zeilennummer und die dazugehörige Anzahl von Sternen im ursprünglichen Muster

Zeilennummer	Anzahl Sterne
0	5
1	4
2	3
3	2
4	1

Die Anzahl der Sterne entspricht den Durchläufen in der inneren Schleife. Sie lässt sich aus der Zeilennummer simpel berechnen

Schreiben Sie das Programm

Problemreduktion (5)

Mögliche Lösung

```
#include<iostream>
using namespace std;
int main()
{
  for (int zeile {0}; zeile < 5; ++zeile) {
    for (int spalte {0}; spalte < 5 - zeile; ++spalte)
      cout << '*';
    cout << '\n';
  }
  return 0;
}
```

Problemlösungsstrategien (1)

Wir haben eine Reihe von Strategien kennengelernt:

Umformulierung: Finde eine äquivalente Formulierung, für die vielleicht schon eine Lösung bekannt ist. Vor allem ein mathematisches Modell ist oft von Vorteil, da die Mathematik schon viele Lösungen anbietet (z.B. kürzester Weg zwischen zwei Städten → Karten als Graph darstellen → Dijkstra Algorithmus)

Divide and conquer: Aufteilen in möglichst unabhängig voneinander zu lösende Teilprobleme

Analogien: Anpassen bereits bekannter Lösungen für analoge Probleme

Problemreduktion: Problem so weit vereinfachen, bis es sich lösen lässt. Eventuell lässt sich die Lösung des einfacheren Problems auf das ursprüngliche Problem verallgemeinern oder anpassen.

Problemlösungsstrategien (2)

Weitere oft anwendbare Strategien:

Wiederverwenden: Bestehender Programmcode kann eventuell für Lösung neuer Probleme verwendet, bzw. adaptiert werden

Experimentieren: Ausprobieren, was unter bestimmten Voraussetzungen passiert, bzw. wie Änderungen am Programm sich auf dessen Verhalten auswirken

Formulieren/Kommunizieren: Versuchen Sie das Problem, das Sie nicht lösen können, jemand anderem mitzuteilen (oft reicht es schon, sich vorzustellen, wie man es jemand anderem erklären könnte).

Jedenfalls sollte man aber immer einen Plan haben, diesen verfolgen und sich nicht frustieren lassen!

Planloses Herumexperimentieren führt nur äußerst selten zum Erfolg

Wenn sich eine der Strategien als nicht erfolgreich herausstellt, so ist das kein Misserfolg. (man hat etwas über das Problem gelernt; vielleicht lässt sich das sogar später woanders einmal sinnvoll einsetzen)

Noch ein Muster

```
*
**
***
****
***
**
*
```

Tipp:
In diesem Beispiel ist es vorteilhaft, auch die Anzahl der Leerzeichen in einer Zeile zu betrachten und negative Zwischenergebnisse zuzulassen
If Anweisungen können auch zum Ziel führen, sind aber nicht notwendig.

Mögliche Funktion

```
*
**
***
****
***
**
*
```

Zeile	*	Differenz (nicht hilfreich)	Anzahl leer (gut, aber es gibt einen Sprung)	3-Zeile (natürliche Folge)	4-|3-Zeile|
0	1	1	3	3	1
1	2	1	2	2	2
2	3	1	1	1	3
3	4	1	0	0	4
4	3	-1	1	-1	3
5	2	-3	2	-2	2
6	1	-5	3	-3	1

universität wien

Mögliche Lösung

```
#include<iostream>
using namespace std;
int main()
{
  for (int zeile {0}; zeile < 7; ++zeile) {
    for (int spalte {0}; spalte < 4-abs(3-zeile); ++spalte)
      cout << '*';
    cout << '\n';
  }
  return 0;
}
```

Rätsel

Bei den folgenden Aufgaben ist zusätzlich auch noch das Ausgabestatement `cout<<' ';` einmal im Programm erlaubt.

a)

```
********
******
****
**
```

b)

```
*              *
**          **
***      ***
********
********
***      ***
**          **
*              *
```

c)

```
*
**
****
********
```

d)

```
**
***
*****
*******
**********
```

Anmerkung: bei den Mustern c und d ist ein völlig anderer Ansatz hilfreich. Beide Muster könnten prinzipiell beliebig nach unten fortgesetzt werden

Überlegen Sie sich eine mögliche Codierung für Muster und schreiben Sie ein Programm, das diese Codierung in ein Muster auf dem Bildschirm umsetzt.

Können Sie eine Funktion schreiben, die Ihre Codierung als Parameter erhält und das Muster ausgibt?

Können Sie sich Muster überlegen, die man nicht mit einem Programm ausgeben kann?