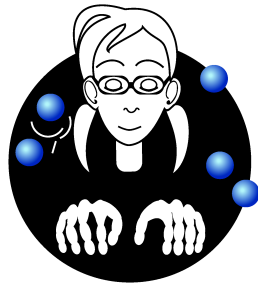
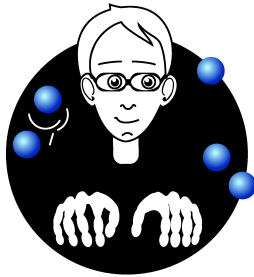


SOI 2009

DIE SCHWEIZER INFORMATIKOLYMPIADE



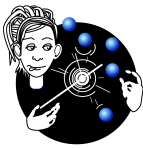


Wie schreibe ich eine gute Lösung?

Bevor wir die Aufgaben präsentieren, möchten wir dir einige Tipps geben, wie eine gute Lösung für die theoretischen Aufgaben aussieht.

Wenn nichts anderes in der Aufgabenstellung geschrieben steht, sollte deine Lösung folgende Punkte enthalten:

- *Quellcode des Programms* in Pascal, C, C++ oder Java. Dein Programm wird nicht durch einen PC getestet, sondern von Hand korrigiert. Darum sollte dein Programm ausführlich kommentiert sein und so elegant und einfach wie möglich sein. Verbringe nicht allzuviel Zeit damit, Eingabe und Ausgabe auf eine originelle Art zu behandeln (es ist nicht verboten, jedoch wird es dir nicht viele zusätzliche Punkte geben; andererseits kann deine Lösung dadurch unverständlicher werden).
- *Beschreibung deiner Lösung*. Das heisst nicht den Quellcode ins Deutsche zu übersetzen. Folgendes sollte enthalten sein:
 - Beschreibung der Grundidee deiner Lösung ohne jegliche Implementationsdetails wie Variablenamen, Funktionen, etc.
 - Beschreibung der benutzten Datenstruktur. Was und in welcher Form müssen wir abspeichern.
 - Beschreibung des Algorithmus. In diesem Abschnitt beschreibst du, unter Benutzung der oben genannten Datenstruktur deine Funktionen, Prozeduren und sonstigen Variablen.
- *Korrektheitsbeweis*. Solange nicht anders angegeben erwarten wir, dass du zeigst, weshalb dein Algorithmus funktioniert. Schau zu, dass du vor allem die nicht offensichtlichen Teile deines Algorithmus darlegst. Zudem solltest du zeigen, dass dein Algorithmus immer terminiert (keine Endlosschleifen, ...).
- *Untersuchen der Laufzeit- und Speicherkomplexität*. Die Laufzeitkomplexität gibt dir eine Abschätzung, wie schnell ein Programm abhängig von



der Grösse der Eingabe läuft. Diese Laufzeit ist proportional zur Anzahl Grundoperationen wie Zuweisungen, Vergleichen von Variablen, arithmetischen Operationen, etc. Wir sind meist an der Laufzeit im schlechtesten Fall interessiert, was bedeutet wie lange dein Algorithmus maximal braucht. Die Speicherkomplexität ist analog dazu – sie gibt eine Abschätzung, wie viel Speicher verbraucht wird in Abhängigkeit der Eingabegrösse.

In der Komplexitätsanalyse sind wir nicht an exakten Grössen interessiert (es ist einerseits zu schwer und andererseits sehr sehr stark von der verwendeten Hardware abhängig). Stattdessen konzentrieren wir uns auf das Wachstum der Komplexität, welche in der O -Notation ausgedrückt wird. Die Funktion $f(n)$ ist in $O(g(n))$, wenn $f(n)$ mal eine Konstante immer kleiner als $g(n)$ ist.¹

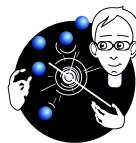
Nehmen wir beispielsweise an, dass ein Programm maximal $T(n) = 0.5n^3 + 5n \log n + 83$ Mikrosekunden für eine Eingabe der Grösse n braucht. Da $0.5n^3$ viel schneller ansteigt als² $5n \log n$, können wir sagen, dass die Laufzeitkomplexität $O(n^3)$ beträgt. (Was wir lesen können als “Die Laufzeit des Algorithmus ist im schlimmsten Fall asymptotisch zu n^3 .”) Wie du sieht, ist die multiplikative Konstante 0.5 dabei unbedeutend. $T(n)$ ist nicht nur in $O(n^3)$, viel mehr ist es auch in $O(n^4)$, $O(n^7)$, ... In der Laufzeitanalyse sind wir daher immer am kleinst möglichen O interessiert, das heisst, an der am langsamsten wachsenden Funktion.

Die Laufzeit- und Speicherkomplexität zu finden ist generell recht einfach. Sobald du zwei verschachtelte Schleifen hast, die beide n mal durchlaufen werden, ergibt das eine Laufzeit von $O(n^2)$. Ein weiteres Beispiel:

¹Mathematisch korrekte Definition:

$f(n) = O(g(n)) \Leftrightarrow \exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+; \forall n > n_0; f(n) \leq c \cdot g(n)$

²Formell, $\lim_{n \rightarrow \infty} \frac{0.5n^3}{5n \log n} = \infty$.



C:

```
int i,j,c;
int A[n];

void main() {
    c = 0;
    for (i=1; i<=n; i++)
        for (j=i+1; j<=n; j++)
            if (A[i]<A[j]) c++;
    printf("%d\n", c);
}
```

Pascal:

```
var i,j,c:integer;
    A:array[1..n] of integer;

begin
    c := 0;
    for i:=1 to n-1 do
        for j:=i+1 to n do
            if A[i]<A[j] then inc(c);
        writeln(c);
    end.
```

Die äussere Schleife wird $n - 1$ mal aufgerufen, die innere $n - i$ mal. Die Vergleichsoperation $A[i] < A[j]$ wird somit $(n - 1) + (n - 2) + \dots + 1 = \frac{n(n-1)}{2}$ mal aufgerufen. Die Laufzeit beträgt somit $O(n^2)$. Wir verwenden 3 Integer Variablen und ein Integer Array der Grösse n . Dadurch ist die Speicherkomplexität $O(n)$.

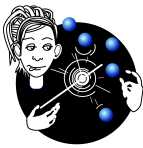
Ein Beispiel

AUFGABE: Dir wird eine Folge von n ganzen Zahlen gegeben. Schreibe ein Programm, das entscheidet, ob die Summe einer zusammenhängenden Teilfolge gleich k ist.

LÖSUNG: Die simpelste Lösung ist alle möglichen Teilfolgen durchzuprobieren und anschliessend ihre Summe mit k zu vergleichen.

```
var n,k:integer;
    C:array[1..MaxN] of integer;
    i,j,z,sum:integer;

begin
    for i:=1 to N do
        for j:=i to N do
            begin
                sum:=0;
                for z:=i to j do sum:=sum+C[z]; { Compute its sum }
```



```
if sum=k then
  begin
    writeln('Subsequence exists'); halt
  end;
end;
writeln('Subsequence does not exist');
end.
```

Dieser Algorithmus ist sehr einfach zu implementieren und der Korrektheitsbeweis sollte auch nicht allzu schwer ausfallen.³ Wie gross ist jedoch die Laufzeitkomplexität? Es ist leicht zu sehen, dass `sum:=sum+C[z]` die am häufigsten aufgerufene Operation ist. Sie wird für jedes Element jeder möglichen Teilfolge aufgerufen. Es gibt $O(n^2)$ Teilfolgen mit je $O(n)$ Elementen. Die Laufzeit beträgt demnach⁴ $O(n^3)$. Geht das auch schneller?

Es ist einfach zu sehen, dass wir die Summe nicht jedes Mal neu berechnen muss. Die Summe $C[i] + \dots + C[j-1] + C[j]$ erhalten wir auch durch hinzufügen des Wertes $C[j]$ zur zuvor berechneten Summe $C[i] + \dots + C[j-1]$. Diese Idee ergibt uns folgendes Programm:

```
begin
  for i:=1 to N do
    sum := 0;
    for j:=i to N do
      begin
        sum := sum + C[j];    { Now sum = C[i]+C[i+1]+...+C[j] }
        if sum=k then
          begin
            writeln('Subsequence exists'); halt
          end;
        end;
      end;
    writeln('Subsequence does not exist');
  end.
```

³Da wir die Summen aller möglichen Teilfolgen berechnen und diese mit k vergleichen.

⁴Dies ist sogar die präziseste Laufzeitabschätzung: Für die genaue Anzahl Operationen erhalten wir

$$\sum_{i=1}^n \sum_{j=i}^n j - i = \frac{n^3 - n}{6}$$



Dieser Algorithmus ist mit $O(n^2)$ Operationen einiges schneller.⁵

Als nächstes werden wir eine kleine Optimierung einbringen: Sobald die Variable `sum` grösser als k wird, beenden wir die innere Schleife und fahren mit dem nächsten Durchlauf der äusseren Schleife fort:

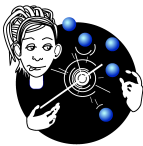
```
begin
  for i:=1 to N do
    sum := 0; j:=i;
    while (sum<k) and (j<=n) do      { End the loop as soon }
      begin                          { as possible }
        sum := sum + C[j];
        if sum=k then
          begin
            writeln('Subsequence exists'); halt
          end;
          j := j+1
        end;
      writeln('Subsequence does not exist');
    end.
```

Leider wurde durch diese Optimierung die Laufzeitkomplexität nicht besser: Für grosse k tut unsere Optimierung nichts. Es gibt jedoch einen anderen Weg, wie wir den Algorithmus verbessern können: Nach dem Beenden der inneren while-Schleife wirft unser Algorithmus den zuvor berechneten Wert für `sum` = $C[i] + C[i+1] + \dots + C[j]$ weg und berechnet $C[i+1] + C[i+2] \dots$ erneut. Da die while-Schleife so schnell als möglich beendet wird, wissen wir, dass `sum` = $C[i] + \dots + C[j] > k$ und $C[i] + \dots + C[j-1] < k$. Somit gilt $C[i+1] + \dots + C[j-1] < k$. Im nächsten Durchlauf der äusseren Schleife können wir für $C[i+1] + \dots + C[j]$ die alte Summe `sum` nehmen, $C[i]$ davon abziehen und damit weiterrechnen:

```
var n,k:integer;                                { Input values }
    C:array[1..MaxN] of integer; { Input sequence }
    i,j,sum:integer;

begin
  sum:=0; i:=1; j:=1;
```

⁵Dies ist wieder die best mögliche Abschätzung der Laufzeit. Die Analyse ist ähnlich wie im Beispiel in der Einführung



```
while (sum<>k) and (i<=n) do    { Check if sum=k }
begin                          { after decrementing }
  while (sum<k) and (j<=n) do { Enlarge the subsequence }
  begin
    sum := sum + C[j];
    if sum=k then
      begin
        writeln('Subsequence exists'); halt
      end;
    j := j+1
  end;
  sum := sum-C[i];            { Shrink the subsequence }
  i := i+1;
end;
if sum=k then writeln('Subsequence exists')
else writeln('Subsequence does not exist');
end.
```

Wie hoch ist nun die Zeitkomplexität dieses Algorithmus? Die äussere Schleife wird maximal n mal aufgerufen. Die innere Schleife kann für jede äussere Schleife $O(n)$ mal aufgerufen werden. Insgesamt kann die innere Schleife jedoch nur n mal aufgerufen werden, da j jedes mal erhöht und nie kleiner wird. Daher ist die Laufzeit dieses Algorithmus $O(n)$. Die Speicherkomplexität bleibt bei $O(n)$. Die Korrektheit des letzten Algorithmus kann folgendermassen bewiesen werden: Wir haben gezeigt, dass der $O(n^3)$ Algorithmus korrekt war. Zudem ist jede unserer Optimierungen zulässig (was wir ebenfalls gezeigt haben).