1. Die bereits gefundenen Primfaktoren befinden sich im Array `faktor`, der noch zu zerlegende Rest in der Variablen `n`. Das Produkt der gefundenen Primfaktoren und `n` muß also die zu zerlegende Zahl `x` ergeben:

$$I : n \cdot \prod_{i=0}^{f-1} \text{faktor}[i] = \text{x} \wedge \forall\, i, 0 \leq i < f : \text{faktor}[i] \text{ ist Primzahl}$$

2. Beweis:

$\{PRE : \Phi = x \in \mathbb{N} \wedge x > 1 \,\forall\, i \in \text{CARD} : \text{primzahl}[i] \text{ ist Primzahl}\}$

$\{0 = 0 \wedge 0 = 0 \wedge x = x \wedge \Phi\}$

`f:= 0; p := 0; n := x;` [Zuweisungsaxiom]

$\{f = 0 \wedge p = 0 \wedge n = x \wedge I \wedge \Phi\}$

`WHILE n # 1 DO` [Regel der Iteration]

$\{I \wedge n \neq 1 \wedge n \text{ div primzahl}[p] = n \text{ div primzahl}[p] \wedge n \text{ mod primzahl}[p] = n \text{ mod primzahl}[p] \wedge \Phi\}$

`quotient := n DIV primzahl[p];` [Zuweisungsaxiom]

`rest := n MOD primzahl[p];` [Zuweisungsaxiom]

$\{I \wedge n \neq 1 \wedge q \cdot \text{primzahl}[p] + r = n \wedge \text{primzahl}[p] \text{ ist Primzahl} \wedge \Phi\}$

`IF rest = 0 THEN` [Regel der Fallunterscheidung]

$\{I \wedge n \neq 1 \wedge q \cdot \text{primzahl}[p] + r = n \wedge r = 0 \wedge \text{primzahl}[p] \text{ ist Primzahl} \wedge \Phi\}$

`faktor[f] := primzahl[p];` [Zuweisungsaxiom]

$\{n \cdot \prod_{i=0}^{f-1} \text{faktor}[i] = x \wedge \forall\, i, 0 \leq i < f : \text{faktor}[i] \text{ ist Primzahl} \wedge n \neq 1 \wedge$

$q \cdot \text{faktor}[f] = n \wedge \text{faktor}[f] \text{ ist Primzahl} \wedge \Phi\}$

$\{q \cdot \prod_{i=0}^{f+1-1} \text{faktor}[i] = x \wedge \forall\, i, 0 \leq i < f : \text{faktor}[i] \text{ ist Primzahl} \wedge n \neq 1 \wedge \Phi\}$

`f := f+1;` [Zuweisungsaxiom]

$\{q \cdot \prod_{i=0}^{f-1} \text{faktor}[i] = x \wedge \forall\, i, 0 \leq i < f : \text{faktor}[i] \text{ ist Primzahl} \wedge n \neq 1 \wedge \Phi\}$

`n := quotient;` [Zuweisungsaxiom]

$\{n \cdot \prod_{i=0}^{f-1} \text{faktor}[i] = x \wedge \forall\, i, 0 \leq i < f : \text{faktor}[i] \text{ ist Primzahl} = I(!) \wedge n \neq 1 \wedge \Phi\}$

`ELSE`

$\{I \wedge n \neq 1 \wedge q \cdot \text{primzahl}[p] + r = n \wedge r \neq 0 \wedge \Phi\}$

`p := p+1;`

$\{I \wedge n \neq 1 \wedge \Phi\}$

`END;`

$\{I \wedge n \neq 1 \wedge \Phi\}$

`END;` $\{I \wedge n = 1 \wedge \Phi\}$

$\{POST : \prod_{i=0}^{f} \text{faktor}[i] = x \wedge \forall\, i, 0 \leq i < f : \text{faktor}[i] \text{ ist Primzahl}$

3. Die Invariante ist eine Aussage, deren Wahrheitswert sich an den 4 Stellen ihrer Gültigkeit nicht ändert. Das ist der invariante Aspekt. Bei jedem Durchlauf einer `WHILE`-Schleife ändert sich jedoch der Bereich, über den die Invariante eine Aussage trifft. So wird z.B. der Bereich der sortierten Zahlen größer oder immer mehr Felder eines Arrays enthalten gültige Werte. Darin besteht der veränderliche Aspekt einer Invariante.

4. Das Hoare-Kalkül basiert auf der Formulierung von Aussagen über den Programmzustand. Es ist deshalb nur für zustandsorientierte Programmiersprachen anwendbar und somit nicht für OPAL.

1. Worst-Case-Aufwand

```
   CONST nMax = 1000;
   TYPE Point = RECORD x,y: INTEGER END;
 1 PROCEDURE ConvexHullWrap(VAR p: ARRAY[0..nMax] OF Point; n: CARDINAL);
 2 VAR i, min, m:      INTEGER;
 3     minangle, v, a: REAL;
 4     t:              Point;
 5 BEGIN
 6 min := 0;

 7 FOR i := 1 TO n-1 DO             (* Finde Punkt mit kleinstem 'y'-Wert (Startpunkt)      *)
 8    IF p[i].y < p[min].y THEN min := i; END;
 9 END;

10 m := -1;
11 p[n] := p[min];                  (* Marke fuer den Abbruch der REPEAT-Schleife:          *)
12 minangle := 0.0;

13 REPEAT                           (* Wiederhole, bis man wieder auf den Startpunkt trifft *)
14    m := m+1;
15    t := p[m]; p[m] := p[min]; p[min] := t;      (* Swap(p[M], p[min])                   *)
16    v := minangle; minangle := 360.0;
17    min := n;

18    FOR i := m+1 TO n DO         (* Suche Punkt für minimalen Winkel                     *)
19        a := Angle(p[m], p[i]);
20        IF (a > v) AND (a < minangle) THEN
21            min := i; minangle := a;
22        END; (* IF *)
23    END; (* FOR *)
24 UNTIL min = n;
25 END ConvexHullWrap;
```

Die Variable, von der der Aufwand im Worst-Case abhängt, ist `N`. Die benötigten Zeiteinheiten jeder Zeile sind in folgender Tabelle gegeben:

Zeile	Aufwand (Ausführungen*Operationen)	Bemerkungen
6	1	
7	$(N-1)\cdot 3 + 1$	1: Initialisierung (Zuweisung), (N-1)*3: Vergleich, Addition, Zuweisung
8	$(N-1)\cdot 2$	
10	1	
11	1	
12	1	
14	$N\cdot 2$	
15	$N\cdot 3$	
16	$N\cdot 2$	
17	N	
18	$(N\cdot(N+1) / 2)\cdot 3 + N\cdot 1$	Punkte, die schon zur Hülle gehören werden nicht noch einmal betrachtet: $\sum_{i=1}^{N}\sum_{j=i}^{N} 1 = \frac{N(N+1)}{2}$
19	$(N\cdot(N+1) / 2)\cdot(2+c)$	Zuweisung, Aufruf der Funktion, c: Operationen der Funktion „Angle"
20	$(N\cdot(N+1) / 2)\cdot 3$	2 Vergleiche, AND
21	$(N\cdot(N+1) / 2)\cdot 2$	
24	$N\cdot 1$	Nach maximal N Iterationen wurde jeder Punkt (und damit auch der Startpunkt) einmal durchlaufen

Zusammen ergibt dies $\left(5+\frac{c}{2}\right)N^2+\left(20+\frac{c}{2}\right)N$ Operationen.

2. Aufwandsklasse

   Da c eine Konstante ist, kann die Polynomregel (Nummer 9) des $\mathcal{O}$-Kalküls angewendet werden:

$$\left(5+\tfrac{c}{2}\right)N^2+\left(20+\tfrac{c}{2}\right)N+1 \overset{\text{Regel 9}}{\Rightarrow} \mathcal{O}\left(N^2\right)$$

   Alternativ können Konstanten k, n_0 angegeben werden, für die $\left(5+\frac{c}{2}\right)N^2+\left(20+\frac{c}{2}\right)N+1 \leq kN^2$ für alle $N > n_0$. Ein Beispiel ist $k = 26 + c$, $n_0 = 1$

3. Dichte Schranke

   Dies wäre genau dann der Fall, wenn „N^2" Element jeder Aufwandsklasse ist, deren Mitglied auch „$\left(5+\frac{c}{2}\right)N^2+\left(20+\frac{c}{2}\right)N+1$" ist:

$$\begin{aligned}
\left(5+\frac{c}{2}\right)N^2+\left(20+\frac{c}{2}\right)N+1 &\leq k_1\,V(N) && ;\ N > n_1\\
\left(5+\frac{c}{2}\right)N^2 &\leq k_1\,V(N)\\
N^2 &\leq \frac{k_1}{5+\frac{c}{2}}\,V(N)\\
N^2 &\leq k_2\,V(N) && ;\ N > n_1
\end{aligned}$$

   Damit ist $n_2 = n_1$ und $k_2 = \frac{k_1}{5+\frac{c}{2}}$.

```
MODULE prog;

FROM InOut IMPORT WriteLn,WriteString;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM Strings IMPORT String;

TYPE
  PTable = POINTER TO TCell;                (* Typ für ganze Tabelle          *)
  TDataType = (TString, TInteger, TFloat);  (* Aufzählungstyp für Datentypen  *)
  TCell = RECORD
    NextRow, NextCol: PTable;
    CASE Type:TDataType OF
      TString:
        TheString: String; |
      TInteger:
        TheInt: INTEGER;   |
      TFloat:
        TheFloat: REAL;
    END;
  END;

PROCEDURE New( Width, Height: INTEGER ): PTable;
(* Legt neue Tabelle der Größe Width x Height an *)
VAR
  i, j: INTEGER;
  first, last, curr, tab: PTable;
BEGIN
  NEW( Tab );               (* Zelle (0, 0)                                    *)
  first := Tab;             (* first zeigt auf das erste Element einer Zeile   *)
  first^.Type := TInteger;
  i := Height;
  REPEAT
    curr := first;          (* curr zeigt auf das aktuelle Element einer Zeile *)
    j := Width;
    REPEAT
      NEW( curr^.NextCol );  (* rechten Nachbarn erzeugen                      *)
      curr := curr^.NextCol; (* rechter Nachbar wird aktuelles Element         *)
      curr^.Type := TInteger;
      DEC( j );
    UNTIL j=0;              (* aktuelle Zeile ist erzeugt                      *)
    NEW( first^.NextRow );  (* unteren Nachbarn erzeugen                       *)
    last := first;
    first := first^.NextRow; (* unterer Nachbar wird neuer akt. Zeilenanfang   *)
    first^.Type := TInteger;
    DEC( i );
  UNTIL i=0;
  DISPOSE( last^.NextRow );  (* es wurde ein Elem. zu viel erzeugt->weg damit  *)
  last^.NextRow := NIL;
  RETURN( Tab );
END New;

PROCEDURE PutString( Tab: PTable; x, y:INTEGER; s:String );
(* speichert s in die Tabelle Tab an Position x, y *)
VAR i: INTEGER;
BEGIN
  FOR i:=1 TO y DO          (* y Zeilen abwärts gehen, ...                   *)
    IF Tab = NIL THEN RETURN;
    ELSE Tab := Tab^.NextRow;
```

```
    END;
  END;
  FOR i:=1 TO x DO
    IF Tab = NIL THEN RETURN;(* ... dann x Spalten nach rechts, ...          *)
    ELSE Tab := Tab^.NextCol;
    END;
  END;
  Tab^.Type := TString;      (* ... dann TCell auf String einstellen und ... *)
  Tab^.TheString := s;       (* ... den String speichern. Fertig.            *)
END PutString;

VAR Tab: TTable;
BEGIN
  Tab := New( 10, 10 );
  PutString( Tab, 3, 5, "Hallo" );
  WriteString( "Fertig" ); WriteLn;
END prog.
```

1.

```
        ORG     0

        LDA     #Feld     ; Pointer initialisieren
        STA     Ptr

next    LDA     Erg       ; Aufsummieren
        ADD     @Ptr
        STA     Erg

        LDA     Ptr       ; Pointer weiterschieben
        ADD     #1
        STA     Ptr

        SUB     #Feld     ; Wenn Ptr-Feld-Anzahl != 0
        SUB     Anzahl
        JNE     next      ; dann sind wir noch nicht fertig

        LDA     Erg       ; Division durchführen
        DIV     Anzahl

        STA     Erg       ; Ergebnis korrekt speichern
        HLT

Ptr     RES     1         ; Datenteil neu definiert
Erg     DAT     0

Anzahl  DAT     5         ; Datenteil (gegeben)
Feld    DAT     -20
        DAT     -4
        DAT     13
        DAT     -10
        DAT     2
END
```

2.

```
t0: MAR <- PC                ; Fetch Operand 1
t1: MBR <- MEM               ;
t2: IR <- MBR, PC <- PC+1    ;

t3: MAR <- IR(O)             ; Decode Operand 1
t4: MBR <- MEM               ;

t5: AC <- MBR                ; Store Operand 1

t6: MAR <- PC                ; Fetch Operand 2
t7: MBR <- MEM               ;
t8: IR <- MBR, PC <- PC+1    ;

t9: MAR <- IR(O)             ; Decode Operand 2
tA: MBR <- MEM               ;

tB: AD <- AC + MBR ; Compute Operation
tC: AC <- AD                 ;

tD: MBR <- AC                ; Save Result
tE: MEM <- MBR               ;
```

3. Die CLU muß um die entsprechenden Mikroprogrammschritte erweitert werden. Die CLU muß ein internes Register erhalten, damit Operation Code sowie Address Mode bis nach Abarbeitung des Mikroprogramms gespeichert werden können, da andernfalls das Nachladen von Operand 2 in das IR augenblicklich dazu

führen würde, daß ein anderes Mikroprogramm ausgeführt wird.

Weiterhin muß ein Bit im Befehlswort für die Signalisierung eines Mehradreßbefehls reserviert werden. Dazu ist eine neue Steuerleitung vom IR zur CLU nötig.

4.
```
        ORG     0

        LDA     Feld            ; Pointer initialisieren
        STA     #Ptr

next    ADDX    @Ptr, Erg       ; Aufsummieren

        ADDX    #1, Ptr         ; Pointer weiterschieben

        SUB     #Feld           ; Wenn Ptr-Feld-Anzahl != 0
        SUB     Anzahl
        JNE     next            ; dann sind wir noch nicht fertig

        DIVX    Erg, Anzahl     ; Division durchführen

STA     Erg             ; Ergebnis korrekt speichern
        HLT

Ptr     RES     1
Erg     DAT     0
Anzahl  DAT     5
Feld    DAT     -20
        DAT     -4
        DAT     13
        DAT     -10
        DAT     2
```

5. Die Variante aus Teil Aufgabenteil 4 ist deutlich schneller, da pro verwendetem Mehradreßbefehl zwei Fetch-Zyklen, also 6 Taktzyklen gespart werden, da separierte LDA und STA entfallen können.

   Zum allgemeinen Vergleich bei direkter Adressierung: der ADDX-Befehl benötigt 14 Taktzyklen, ein LDA plus ADD benötigen bereits 13 Zyklen, ein STA nochmals 6 Zyklen. Der Geschwindigkeitsvorteil hängt ganz Allgemein von der Unterschiedlichkeit der Länge der Mikroprogramme ab – durch das Sparen der Fetch-Zyklen ist der ADDX schneller, wenn sonst die Kombination LDA, ADD, STA verwendet worden wäre.

   Falls ein STA nicht nötig gewesen wäre, ist der ADDX der normalen Addition unterlegen, da der ADDX automatisch die Speicherung miterledigt. Im Falle, daß sogar eine Sequenz von Operationen ausgeführt wird (LDA, ADD, MUL, ADD, MUL, ..., STA) ist der ADDX-Befehl sogar deutlich langsamer, da die von ihm durchgeführte Speicherung des Ergebnisses unerwünscht ist. Er benötigt obendrein noch eine zusätzliche Variable zur Speicherung des Zwischenergebnisses, während in einer konventionellen Rechenkette das Ergebnis ohne Speicherzugriff im Akkumulator gelagert wird.