Abgabetermin: Sa in der KW 41

☐ Gr. 1, DI Franz Gruber-Leitner
☒ Gr. 2, Dr. Erik Pitzer

Name Roman Lumetsberger

Aufwand in h 9

Punkte ________ Kurzzeichen Tutor / Übungsleiter ______ / ______

1. Krafttraining (7 Punkte)

Ein Freund von Ihnen beschließt, nachdem er diesen Sommer nicht viel Zeit gehabt hat sich im Freien sportlich zu betätigen, ab jetzt regelmäßig das Fitnessstudio zu besuchen. Dort trainiert er mit einer Langhantel und verschiedenen Gewichten. Es fällt ihm allerdings ziemlich schwer für ein bestimmtes Gewicht, die richtige Kombination an Hantelscheiben zu finden. Da Sie sich vor kurzem mit dem Backtracking-Verfahren auseinandergesetzt haben, schlagen Sie vor ihm dabei zu helfen.

Schreiben Sie ein Programm `weights`, das als Parameter ein bestimmtes Gewicht bekommt und die richtige Kombination an Hantelscheiben ausgibt.

Die Gewichte und die Anzahl jeder verfügbaren Scheibe, sowie das Gewicht der Hantelstange selbst, können dazu in zwei globalen Feldern hartcodiert werden, wobei jeweils die Anzahl von Paaren von Scheiben angegeben wird, z.B.:

```
const double weights[] = { 0.5, 1.25, 2.5, 5, 10, 15, 20, 25 };
const int counts[]     = {   1,    1,   1, 3,  1,  1,  3,  1 };
const double bar = 20;
```

Falls es nicht möglich ist, das gewünschte Gewicht mit den verfügbaren Scheiben zu erreichen soll eine dementsprechende Meldung ausgegeben werden.

2. Mathematik mit Polynomen und Divide&Conquer (2 + 2 + 3 + 3 + 7 Punkte)

In der Mathematik könnte man sagen: „Fast alles ist ein Polynom“. Das beginnt bei den Zahlen eines beliebigen Zahlensystems und geht bis hin zu allen Funktionen, die entweder schon Polynome sind oder durch Interpolation mittels Polynomen näherungsweise dargestellt werden können. Deshalb wird es höchste Zeit, sich auch in der Programmierung mit Polynomen zu beschäftigen.

Ein Polynom p des Grades n hat die Form

$$p(x) = p_0 + p_1x + p_2x^2 + p_3x^3 + \cdots + p_nx^n$$

und kann softwaretechnisch durch ein Feld, das die n+1 Koeffizienten p_0 bis p_n enthält, dargestellt werden.

Entwickeln Sie ein C-Programm, das Berechnungen mit Polynomen durchführt, und dazu folgende Funktionen zur Verfügung stellt:

a) Eine (Hilfs-)Funktion, die ein Polynom p in der üblichen Form (s.o.) anzeigt. Schnittstelle:

```
void printPoly(int n, double p[]);
```

Beispiel: Für $p(x) = 1 + x + 3x^2 - 4x^3$ soll `printPoly` folgendes ausgeben:

```
1 + 1x + 3x^2 - 4x^3
```

Tipp: Für die kompakte Formatierung von Fließkommazahlen können sie das Format `"%g"` verwenden.

b) Eine Funktion, die ein Polynom p an der Stelle x auswertet, also $p(x)$ berechnet und dafür natürlich das Horner-Schema verwendet. Schnittstelle:

```
double evalPoly(int n, double p[], double x);
```

c) Eine Funktion, die die Summe zweier Polynome p und q bildet, indem deren Koeffizienten paarweise addiert werden. Schnittstelle:

```
int polySum(int np, double p[], int nq, double q[], double r[]);
```

Diese Funktion liefert im Ausgabeparameter r die Summe von p und q. Das Funktionsergebnis ist der Grad von r, also das Maximum der Grade von p und q, also $\max(np, nq)$, z.B.:

$$p(x) = 1 + x + 3x^2 - 4x^3$$
$$q(x) = 1 + 2x - 5x^2 - 3x^3$$
$$p(x) + q(x) = r(x) = 2 + 3x - 2x^2 - 7x^3$$

d) Eine Funktion, die das Produkt zweier Polynome p und q bildet, indem deren Koeffizienten so miteinander multipliziert werden, dass jeder Koeffizient des zweiten Polynoms mit jedem des ersten multipliziert wird. Sind beide vom Grad n, so erfordert dies $(n+1)^2$ Multiplikationen, führt also zu einem $O(n^2)$-Verfahren (bezogen auf die Anzahl der Multiplikationen):

```
int polyProd(int np, double p[], int nq, double q[], double r[]);
```

Diese Funktion liefert im Ausgabeparameter r das Produkt von p und q. Das Funktionsergebnis ist der Grad von r, also das die Summe der Grade von p und q, also $np + nq$, z.B.:

$$p(x) = 1 + x + 3x^2 - 4x^3$$
$$q(x) = 1 + 2x - 5x^2 - 3x^3$$
$$p(x) \cdot q(x) = r(x) = 1 + 3x - 6x^3 - 26x^4 + 11x^5 + 12x^6$$

e) Die Krönung Ihrer Implementierung ist eine Funktion zur Bildung des Produkts, die durch Anwendung des Divide&Conquer-Prinzips nun aber ein $O(n^{1.58})$-Verfahren implementiert:

```
int polyProdFast(int np, double p[], int nq, double q[], double r[]);
```

Ohne Beschränkung der Allgemeinheit können Sie annehmen (*und dürfen es auch so implementieren!*), dass

- beide Polynome (p und q) vom selben Grad n sind ($n = n_p = n_q$),
- und dass $n+1$ eine Zweierpotenz ist ($n + 1 = 2k$ mit $k > 1$).

Idee des Verfahrens: Beide Polynome werden in der Mitte in ein niederwertiges (*low*, *l*) und ein höherwertiges (*high*, *h*) Polynom zerlegt, also z.B. für $p(x)$

$$p_l(x) = p_0 + p_1 x + \cdots + p_{n/2} x^{n/2}$$
$$p_h(x) = p_{n/2+1} + p_{n/2+2} x + \cdots + p_n x^{n/2}$$

Damit kann das Produkt auch wie folgt berechnet werden:

$$p \cdot q = p_l \cdot q_l + (p_l \cdot q_h + q_l \cdot p_h) \cdot x^{n/2+1} + p_h \cdot q_h \cdot x^{x+1}$$

Das bringt allerdings noch keinen Vorteil, denn es sind nun vier Multiplikationen von Polynomen halber Länge notwendig, und weil $4 \cdot \left(\frac{n+1}{2}\right)^2 = (n+1)^2$ ist, handelt es sich immer noch um ein quadratisches Verfahren.

Eine Verbesserung ergibt sich erst, wenn man ganz genau hinschaut und erkennt, dass man, wenn man drei Hilfspolynome r_l, r_h und r_m definiert, das Produkt, auch folgendermaßen berechnen kann:

$$r_l = p_l \cdot q_l$$
$$r_h = p_h \cdot q_h$$
$$r_m = (p_l + p_h) \cdot (q_l + q_h)$$
$$p \cdot q = r_l + (r_m - r_l - r_h) \cdot x^{n/2+1} + r_h \cdot x^{n+1}$$

Damit sind „nur“ noch drei Polynommultiplikationen (je eine für r_l, r_h und r_m) notwendig. Das „Zusammenbauen“ zum Schluss erfolgt einfach durch Zuweisung der Elemente von r_l, r_h und r_m an die richtigen Indizes von r.

Beispiel

$$p(x) = 1 + x + 3x^2 - 4x^3$$
$$q(x) = 1 + 2x - 5x^2 - 3x^3$$

Diese Polynome lassen sich nun folgendermaßen zerlegen:

$$p_l(x) = 1 + x \qquad q_l(x) = 1 + 2x$$
$$p_h(x) = 3 - 4x \qquad q_h(x) = -5 - 3x$$

Daraus ergibt sich als Zwischenrechnungen:

$$r_l(x) = (1 + x) \cdot (1 + 2x) = 1 + 3x + 2x^2$$
$$r_h(x) = (3 - 4x) \cdot (-5 - 3x) = -15 + 11x + 12x^2$$
$$r_m(x) = (4 - 3x) \cdot (-4 - x) = -16 + 8x + 3x^2$$
$$r_m(x) - r_l(x) - r_h(x) = -2 - 6x - 11x^2$$

Schließlich kommen wir wieder zu demselben Produkt wie schon im Beispiel zu Punkt d):

$$r(x) = 1 + 3x - 6x^3 - 26x^4 + 11x^5 + 12x^6$$

Hinweise:

1. Für die Implementierung dieser Aufgabe ist *viel weniger Mathematik* notwendig als auf den ersten Blick ersichtlich. Sie müssen lediglich mit Feldern und Indizes jonglieren.
2. Sie können diese Übung komplett ohne dynamischen Speicher implementieren, indem Sie die Felder für das Resultat groß genug wählen und bereits am Stack anlegen. Falls Sie dennoch bereits mit dynamisch allokiertem Speicher arbeiten wollen (nicht empfohlen!), überlegen und dokumentieren Sie genau, wer für das Reservieren und Freigeben verantwortlich ist und stellen Sie sicher, dass keine Speicherlecks entstehen.

Allgemeine Hinweise

1. Geben Sie für alle Ihre Lösungen immer eine **Lösungsidee** an.
2. **Kommentieren** und **testen** Sie Ihre Programme **ausführlich**.

1 Aufgabe 1 - Krafttraining

1.1 Lösungsidee

Laut Angabe werden die benötigten Felder als globale Variablen angelegt. Dabei ist die Anzahl der Scheiben **immer ein Scheibenpaar**. Das heißt, dass immer die doppelte Anzahl an Scheiben vorhanden ist.
Bsp.:

```
weights[0] = 0.5;
counts[0] = 1; /* 2 Scheiben mit 0.5 kg = 1 kg*/
```

Dieses Programm erfordert einen Eingabeparameter, der geprüft und auf double konvertiert werden muss.
Da das Gewicht der Langhantel in dem eingegebenen Gewicht enthalten ist, muss dieses abgezogen werden. Weiters muss das Gewicht dann durch 2 dividiert werden, da die Gewichtscheiben in **Paaren** angegeben sind. Durch die Division durch 2 wird dann genau 1 Seite der Hantel berechnet.
Dieses Beispiel ist, wie in der Angabe erwähnt, mit dem Backtracking-Verfahren zu lösen. Dabei werden alle möglichen Gewichte rekursiv durchlaufen. Es wird dann für jede verfügbare Anzahl von Gewichten geprüft, ob das aktuelle Gewicht plus die neuen Scheiben kleiner gleich dem gewünschten Gewicht ist.

- Ist dies der Fall, dann kann mit dem nächsten Gewicht weitergemacht werden.
- Ist dies nicht der Fall, dann kann die aktuelle Teillösung verworfen werden.

Wurden alle Gewichte durchlaufen, kann geprüft werden, ob das gewünschte Gewicht erreicht ist. Wenn ja, dann wurde eine mögliche Lösung gefunden und kann ausgegeben werden.

1.2 Sourcecode

```c
/*****************************************************************************
  weight.c
  Roman Lumetsberger

  Implements the backtrack algorithm for finding the correct count of weight-pairs
  for a given weight.
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <errno.h>
#include <math.h>

/* defines the length of the input array */
#define ARRAY_LENGTH 8

/* boolean type */
typedef enum bool {false,true} bool;

/* global definition of weights */
const double weights[] = { 0.5, 1.25, 2.5, 5, 10, 15, 20, 25 };
/*global definitions of pairs per weight */
const int counts[] = { 1, 1, 1, 3, 1, 1, 3, 1 };
/* weight of the bar */
const double bar = 20;

/* array for used weights */
int usedWeights[ARRAY_LENGTH];

/* compares 2 double values against the defined epsilon */
bool compareDouble(double left, double right) {
  return fabs(left - right) < DBL_EPSILON;
}

/* prints the solution */
void printSolution() {
  int i;
  printf("Lösung:\n");
  printf("%6.2f kg : Hantelstange\n", bar);
  for(i = 0; i < ARRAY_LENGTH; i++) {
    if(usedWeights[i] > 0)
      printf("%6.2f kg : %-d Paar(e)\n", weights[i], usedWeights[i]);
  }
}

```

```c
/* recursive backtrack function */
bool addWeight(int index, double neededWeight) {
  static double currentWeight;
  int i;
  double tmpWeight;

  /* reset currentWeight on first index*/
  if ( index == 0 )
    currentWeight = 0;

  for(i = 0; i <= counts[index]; i++) {
    tmpWeight = weights[index] * i;
    /* backtrack step, check if the new weight is <= needed weight*/
    if((currentWeight + tmpWeight - neededWeight) < DBL_EPSILON) {
      usedWeights[index] = i;
      currentWeight += tmpWeight;
      /* check if it is the last weight */
      if( index == ARRAY_LENGTH - 1 ) {
        /*solution found? */
        if(compareDouble(currentWeight, neededWeight)) {
          printSolution(ARRAY_LENGTH, usedWeights);
          return true;
        }
      }
      else {
        /* add next weight and stop if a solution is found */
        if(addWeight(index + 1, neededWeight))
          return true;
      }
      /* remove the weight again */
      currentWeight -= tmpWeight;
    }
  }
  return false;
}

int main(int argc, char **argv) {
  char *endptr;
  double inputWeight;

  if(argc != 2 ) {
    printf("Ungültige Parameter\n");
    printf("Usage %s Gewicht\n", argv[0]);
    return EXIT_FAILURE;
  }

  endptr = NULL;
  errno = 0;
  inputWeight = strtod(argv[1], &endptr);
```

```c
  /* Check for errors */
  if ((errno == ERANGE && (inputWeight == HUGE_VAL || inputWeight == 0))
      || (inputWeight == 0 && errno != 0)) {
    printf("Gewicht ungültig, bitte Zahl größer 0 eingeben\n");
    return EXIT_FAILURE;
  }

  /* complete or partial string was invalid */
  if (endptr == argv[1] || *endptr != '\0') {
    printf("Gewicht ungültig, bitte Zahl größer 0 eingeben\n");
    return EXIT_FAILURE;
  }

  if(inputWeight <= 0) {
    printf("Gewicht ungültig, bitte Zahl größer 0 eingeben\n");
    return EXIT_FAILURE;
  }

  /* remove the weight of the bar and devide by 2 to calcualte one side */
  inputWeight = (inputWeight - bar) / 2;
  /* start with first weight */
  if(!addWeight(0, inputWeight  )) {
      printf("Keine Lösung\n");
  }
  return EXIT_SUCCESS;
}
```

1.3 Testfälle

1.3.1 Testfall 1 - ungültige Parameter

Erwartetes Ergebnis: Fehlermeldung

```
romanlum@ubuntu: ~/swo3/UebungMoodle2
romanlum@ubuntu:~/swo3/UebungMoodle2$ ./weight
Ungültige Parameter
Usage ./weight Gewicht
romanlum@ubuntu:~/swo3/UebungMoodle2$ ./weight a
Gewicht ungültig, bitte Zahl größer 0 eingeben
romanlum@ubuntu:~/swo3/UebungMoodle2$ ./weight a1
Gewicht ungültig, bitte Zahl größer 0 eingeben
romanlum@ubuntu:~/swo3/UebungMoodle2$ ./weight 1a
Gewicht ungültig, bitte Zahl größer 0 eingeben
romanlum@ubuntu:~/swo3/UebungMoodle2$ ./weight hallo
Gewicht ungültig, bitte Zahl größer 0 eingeben
romanlum@ubuntu:~/swo3/UebungMoodle2$ ./weight zwei
Gewicht ungültig, bitte Zahl größer 0 eingeben
```

1.3.2 Testfall 2 - Lösung ohne Scheiben

Eingabe: 20
Erwartetes Ergebnis: nur Hantelstange

```
romanlum@ubuntu: ~/swo3/UebungMoodle2
romanlum@ubuntu:~/swo3/UebungMoodle2$ ./weight 20
Lösung:
 20.00 kg : Hantelstange
romanlum@ubuntu:~/swo3/UebungMoodle2$
```

1.3.3 Testfall 3 - Lösung 1 Paar

Eingabe: 21, 70
Erwartetes Ergebnis: Hantelstange + 1 Paar

```
romanlum@ubuntu: ~/swo3/UebungMoodle2/aufgabe1
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe1$ ./weight 21
Lösung:
 20.00 kg : Hantelstange
  0.50 kg : 1 Paar(e)
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe1$ ./weight 70
Lösung:
 20.00 kg : Hantelstange
 25.00 kg : 1 Paar(e)
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe1$
```

1.3.4 Testfall 4 - Lösung mit mehreren Paaren

Eingabe: 140, 82.5, 115
Erwartetes Ergebnis: Hantelstange + mehrere Paare

```
romanlum@ubuntu: ~/swo3/UebungMoodle2/aufgabe1
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe1$ ./weight 140
Lösung:
 20.00 kg : Hantelstange
 20.00 kg : 3 Paar(e)
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe1$ ./weight 82.5
Lösung:
 20.00 kg : Hantelstange
  1.25 kg : 1 Paar(e)
 10.00 kg : 1 Paar(e)
 20.00 kg : 1 Paar(e)
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe1$ ./weight 115
Lösung:
 20.00 kg : Hantelstange
  2.50 kg : 1 Paar(e)
 20.00 kg : 1 Paar(e)
 25.00 kg : 1 Paar(e)
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe1$
```

1.3.5 Testfall 5 - Lösung mit allen Paaren

Eingabe: 278,5
Erwartetes Ergebnis: Hantelstange + alle Paaren

```
romanlum@ubuntu: ~/swo3/UebungMoodle2/aufgabe1
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe1$ ./weight 278.5
Lösung:
 20.00 kg : Hantelstange
  0.50 kg : 1 Paar(e)
  1.25 kg : 1 Paar(e)
  2.50 kg : 1 Paar(e)
  5.00 kg : 3 Paar(e)
 10.00 kg : 1 Paar(e)
 15.00 kg : 1 Paar(e)
 20.00 kg : 3 Paar(e)
 25.00 kg : 1 Paar(e)
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe1$
```

1.3.6 Testfall 6 - keine Lösung

Eingabe: 10, 22, 27, 10000, 1000, 20.1, 44.45
Erwartetes Ergebnis: keine Lösung

```
romanlum@ubuntu: ~/swo3/UebungMoodle2
romanlum@ubuntu:~/swo3/UebungMoodle2$ ./weight 10
Keine Lösung
romanlum@ubuntu:~/swo3/UebungMoodle2$ ./weight 22
Keine Lösung
romanlum@ubuntu:~/swo3/UebungMoodle2$ ./weight 27
Keine Lösung
romanlum@ubuntu:~/swo3/UebungMoodle2$ ./weight 10000
Keine Lösung
romanlum@ubuntu:~/swo3/UebungMoodle2$ ./weight 1000
Keine Lösung
romanlum@ubuntu:~/swo3/UebungMoodle2$ ./weight 20.1
Keine Lösung
romanlum@ubuntu:~/swo3/UebungMoodle2$ ./weight 44.45
Keine Lösung
romanlum@ubuntu:~/swo3/UebungMoodle2$
```

2 Aufgabe 2 - Mathematik mit Polynomen und Divide & Conquer

2.1 Lösungsidee

Bei dieser Aufgabe müssen die angegebenen Funktionen implementiert werden, dadurch werden die Testfälle direkt im Code erzeugt und können mit einem Kommandozeilenparameter ausgeführt werden.

2.1.1 printPoly

Diese Funktion gibt ein Polynom, das durch den Grad und das Feld der Koeffizienten definiert ist, auf dem Bildschirm aus. Dabei muss das Feld in einer Schleife durchlaufen werden und die einzelnen Koeffizienten und deren Potenz ausgegeben werden.
Dabei definiert der Inhalt des Feldes den Koeffizienten und der Index die Potenz von x. Null Werte dürfen nicht ausgegeben werden.

2.1.2 evalPoly

Diese Funktion berechnet ein Polynom für einen bestimmten Wert x. Wie in der Angabe erwähnt, soll das Horner Schema für die Berechnung herangezogen werden. Dies lässt sich durch eine Schleife, die von hinten nach vorne läuft, umsetzen. Dabei wird immer der aktuelle Wert des Feldes an der betrachteten Stelle zum Ergebnis addiert und anschließend mit x multipliziert. Ist der Index = 0, darf nicht mehr mit x multipliziert werden.
Bsp.:$(((0 + 3) * x + 4) * x + 5) * x + 3$

2.1.3 polySum

Diese Funktion berechnet die Summe zweier Polynome. Dabei muss zu Beginn das Maximum der Grade der Polynome bestimmt werden. Eine Schleife läuft dann von 0 bis zum berechneten Grad. Die Summe der Werte an der aktuellen Position wird in das Ergebnisfeld geschrieben. Dabei muss darauf geachtet werden, dass nicht über die Grenzen der einzelnen Polynome hinausgelaufen wird.
Der berechnete Grad ist der Rückgabewert der Funktion.

2.1.4 polyProd

Diese Funktion berechnet das Produkt zweier Polynome. Dabei werden 2 verschachtelte Schleifen benötigt. Die erste läuft von 0 bis zur Länge des ersten Feldes. Die zweite auch wieder von 0 bis zur Länge des zweiten Feldes.
In das Ergebnisfeld wird dann das Produkt von des Koeffizienten des ersten Polynomes multipliziert mit dem Koeffizienten des zweiten Polynoms gespeichert. Dabei ist der Index die Summe der beiden Laufvariablen, da bei einer Multiplikation sich ja auch die Potenz ändert.
Die Summe der beiden Grade ist der Rückgabewert der Funktion.

2.1.5 polyProdFast

Diese Funktion berechnet ebenfalls das Produkt zweier Polynome. Hier müssen laut Angabe die Polynome gleichen Grades und der Grad $+ 1$ muss eine 2er Potenz sein. Die Polynome müssen dann geteilt werden und dann rl, rh und rm laut Formel in der Angabe berechnet werden. Dabei

wird für die Multiplikationen wieder die Methode rekursiv aufgerufen. Die Hilfspolynome rl, rh und rm können dann in einer Schleife zu dem Ergebnis r zusammengefügt werden. Dabei kann $x^{n/2+1}$ und x^{n+1} durch verschieben des Indexes erreicht werden.
Die Abbruchbedingung der Rekursion ist, wenn der Grad = 0 ist, denn dann kann die skalare Multiplikation verwendet werden.
Rückgabewert:

- Ist der Grad größer Null, dann ist der Rückgabewert die Summe der Grade.
- Ist der Grad gleich Null, dann ist dieser gleich 1.

2.2 Sourcecode

```c
/*****************************************************************************
  poly.c
  Roman Lumetsberger

  Implements mathematical functions for polynomials
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <string.h>

#define MAX_ARRAY_SIZE 255

/* prints the polynomial, n is the degree,
   array size has to be n + 1 */
void printPoly(int n, double p[]) {
  int i;

  if(n < 0 )
    return;

  /* print first factor */
  printf("%g", p[0]);

  for(i = 1; i <= n; i++) {
    /* do not print 0 values */
    if( p[i] != 0 ) {
      if( p[i] < 0 )
        printf(" - ");
      else
        printf(" + ");

      /* print the absolute value, because +/- is alredy printed */
      printf("%gx", fabs(p[i]));

```

```
      if( i > 1 )
        printf("^%d", i);
    }
  }
  printf("\n");
}

/* evaluates the polynomial, n is the degree
  array size has to be n+1 */
double evalPoly(int n, double p[], double x) {
  int i;
  double result = 0;

  for(i = n; i >= 0; i--) {
    result += p[i];
    if(i > 0)
      result = result * x;
  }
  return result;
}

/* calculates the sum of p and q, np and nq are the degree,
   the array size has to be np+1 and nq+1 */
int polySum(int np, double p[], int nq, double q[], double r[]) {
  int factor;
  int i;

  if( np > nq )
    factor = np;
  else
    factor = nq;

  for( i = 0; i <= factor; i++) {
    if ( i <= np )
      r[i] = p[i];
    else
      r[i] = 0;

    if(i <= nq )
      r[i] += q[i];
  }
  return factor;
}

/* calculates the product of p and q, np and nq are the degree,
   the array size has to be np+1 and nq+1 */
int polyProd(int np, double p[], int nq, double q[], double r[]) {
  int i, j;
  for(i = 0; i <= np; i++) {
```

```c
    for(j = 0; j <= nq; j++) {
      r[i + j] += p[i] * q[j];
    }
  }
  return np + nq;
}

/* calculates the product of p and q, np and nq are the degree,
   the array size has to be np+1 and nq+1 */
int polyProdFast(int np, double p[], int nq, double q[], double r[]) {
  double rl[MAX_ARRAY_SIZE] = {0};
  double rh[MAX_ARRAY_SIZE] = {0};
  double rm[MAX_ARRAY_SIZE] = {0};
  double plph[MAX_ARRAY_SIZE] = {0};
  double qlqh[MAX_ARRAY_SIZE] = {0};
  int i, splitLevel;

  /* assert the given values */
  assert(np == nq);

  /* use scalar multiplication for degree = 0*/
  if( np == 0 ) {
    r[0] = p[0] * q[0];
    return 1;
  }

  /* factor is the splited level
    np is the degree of the polynomial
    that's why we have to use np + 1 here
  */
  splitLevel = (int) (np + 1) / 2;

  /* check the max array size */
  assert(splitLevel < MAX_ARRAY_SIZE);

  /* calculate the sub polynomials */
  for(i = 0; i < splitLevel; i++) {
    plph[i] = p[i] + p[i + splitLevel];
    qlqh[i] = q[i] + q[i + splitLevel];
  }

  /* calculate the product of the sub polynoms
   splitLevel -1 is used for getting pl and ql */
  polyProdFast(splitLevel -1, p, splitLevel -1, q, rl);
  /* ph, qh is p,q shifted of split level */
  polyProdFast(splitLevel -1, p + splitLevel, splitLevel -1, q + splitLevel, rh);
  polyProdFast(splitLevel -1, plph, splitLevel -1, qlqh, rm);

  /* combine the results */
```

```
  for(i = 0; i <= np; i++) {
    r[i] += rl[i];
    r[i + splitLevel] += rm[i] - rl[i] - rh[i];
    r[i + np + 1] += rh[i];
  }
  return np + nq;
}


int main(int argc, char **argv) {
  double p[] = {1, 1, 3, -4, 6.4, 0, 2, 6, 2, 2, 3, 4, 5, 6, -7 ,3, 3.3};
  double q[] = {1, 2, -5, -3, 9.98, 5.98, 4, 9, 2, -4, -6, -9, -7, 9.9, 8.33, 16, 4.4};
  double r[MAX_ARRAY_SIZE] = {0};
  int testcase, i, newDegree;
  double result;

  if(argc != 2) {
    printf("Falsche Eingabe!\n");
    printf("Usage: %s Testfall\n", argv[0]);
    return EXIT_FAILURE;
  }

  testcase = atoi(argv[1]);

  /* coded test cases */
  switch(testcase) {
    case 1:
      printf("Testfall 1 - printPoly von  0 - 15 \n");
      for(i = 0; i < 16; i++) {
        printPoly(i, p);
      }
      break;

    case 2:
      printf("Testfall 2 - printPoly von  0 - 15 \n");
      for(i = 0; i < 16; i++) {
        printPoly(i, q);
      }
      break;

    case 3:
      printf("Testfall 3 - evalPoly von  0 - 5 \n");
      printf("Test-Polynom: ");
      printPoly(8, p);
      for(i = 0; i < 6; i++) {
        result = evalPoly(8, p, i);
        printf("Ergebnis bei x = %d: %g\n", i, result);
      }
      break;
```

```

    case 4:
      printf("Testfall 4 - evalPoly von  0 - 5 \n");
      printf("Test-Polynom: ");
      printPoly(3, q);
      for(i = 0; i < 6; i++) {
        result = evalPoly(3, q, i);
        printf("Ergebnis bei x = %d: %g\n", i, result);
      }
      break;

    case 5:
      printf("Testfall 5 - polySum von  p+q \n");
      printf("Test-Polynom p: ");
      printPoly(3, p);
      printf("Test-Polynom q: ");
      printPoly(3, q);
      newDegree = polySum(3, p, 3, q, r);
      printf("Result (%d): ", newDegree);
      printPoly(newDegree, r);
      break;

    case 6:
      printf("Testfall 6 - polySum von  p+q \n");
      printf("\nTest-Polynom p: ");
      printPoly(2, p);
      printf("Test-Polynom q: ");
      printPoly(3, q);
      newDegree = polySum(2, p, 3, q, r);
      printf("Result (%d): ", newDegree);
      printPoly(newDegree, r);
      break;

    case 7:
      printf("Testfall 7 - polySum von  p+q \n");
      printf("\nTest-Polynom p: ");
      printPoly(3, p);
      printf("Test-Polynom q: ");
      printPoly(2, q);
      newDegree = polySum(3, p, 2, q, r);
      printf("Result (%d): ", newDegree);
      printPoly(newDegree, r);
      break;

     case 8:
      printf("Testfall 8 - polyProd von  p und q \n");
      printf("Test-Polynom p: ");
      printPoly(3, p);
      printf("Test-Polynom q: ");
```

```
      printPoly(3, q);
      newDegree = polyProd(3, p, 3, q, r);
      printf("Result (%d): ", newDegree);
      printPoly(newDegree, r);
      break;

    case 9:
      printf("Testfall 9 - polyProd von  p und q \n");
      printf("\nTest-Polynom p: ");
      printPoly(2, p);
      printf("Test-Polynom q: ");
      printPoly(3, q);
      newDegree = polyProd(2, p, 3, q, r);
      printf("Result (%d): ", newDegree);
      printPoly(newDegree, r);
      break;

    case 10:
      printf("Testfall 10 - polyProd von  p und q \n");
      printf("\nTest-Polynom p: ");
      printPoly(3, p);
      printf("Test-Polynom q: ");
      printPoly(2, q);
      newDegree = polyProd(3, p, 2, q, r);
      printf("Result (%d): ", newDegree);
      printPoly(newDegree, r);
      break;

    case 11:
      printf("Testfall 11 - polyProdFast von  p und q \n");
      for(i = 4; i < 9; i = i * 2) {
        printf("\nTest-Polynom p: ");
        printPoly(i-1, p);
        printf("Test-Polynom q: ");
        printPoly(i-1, q);
        newDegree = polyProdFast(i-1, p, i-1, q, r);
        printf("Result (%d): ", newDegree);
        printPoly(newDegree, r);
        /* clear r again */
        memset(r, 0, sizeof(r));
      }
      break;

    default: printf("Ungültiger Testfall\n");
  }

  return EXIT_SUCCESS;
}
```

2.3 Testfälle

2.3.1 Testfall 1 - printPoly

romanlum@ubuntu: ~/swo3/UebungMoodle2/aufgabe2

```
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$ ./poly 1
Testfall 1 - printPoly von  0 - 15
1
1 + 1x
1 + 1x + 3x^2
1 + 1x + 3x^2 - 4x^3
1 + 1x + 3x^2 - 4x^3 + 6.4x^4
1 + 1x + 3x^2 - 4x^3 + 6.4x^4
1 + 1x + 3x^2 - 4x^3 + 6.4x^4 + 2x^6
1 + 1x + 3x^2 - 4x^3 + 6.4x^4 + 2x^6 + 6x^7
1 + 1x + 3x^2 - 4x^3 + 6.4x^4 + 2x^6 + 6x^7 + 2x^8
1 + 1x + 3x^2 - 4x^3 + 6.4x^4 + 2x^6 + 6x^7 + 2x^8 + 2x^9
1 + 1x + 3x^2 - 4x^3 + 6.4x^4 + 2x^6 + 6x^7 + 2x^8 + 2x^9 + 3x^10
1 + 1x + 3x^2 - 4x^3 + 6.4x^4 + 2x^6 + 6x^7 + 2x^8 + 2x^9 + 3x^10 + 4x^11
1 + 1x + 3x^2 - 4x^3 + 6.4x^4 + 2x^6 + 6x^7 + 2x^8 + 2x^9 + 3x^10 + 4x^11 + 5x^12
1 + 1x + 3x^2 - 4x^3 + 6.4x^4 + 2x^6 + 6x^7 + 2x^8 + 2x^9 + 3x^10 + 4x^11 + 5x^12 + 6x^13
1 + 1x + 3x^2 - 4x^3 + 6.4x^4 + 2x^6 + 6x^7 + 2x^8 + 2x^9 + 3x^10 + 4x^11 + 5x^12 + 6x^13 - 7x^14
1 + 1x + 3x^2 - 4x^3 + 6.4x^4 + 2x^6 + 6x^7 + 2x^8 + 2x^9 + 3x^10 + 4x^11 + 5x^12 + 6x^13 - 7x^14 + 3x^15
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$
```

2.3.2 Testfall 2 - printPoly - Polynom 2

romanlum@ubuntu: ~/swo3/UebungMoodle2/aufgabe2

```
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$ ./poly 2
Testfall 2 - printPoly von  0 - 15
1
1 + 2x
1 + 2x - 5x^2
1 + 2x - 5x^2 - 3x^3
1 + 2x - 5x^2 - 3x^3 + 9.98x^4
1 + 2x - 5x^2 - 3x^3 + 9.98x^4 + 5.98x^5
1 + 2x - 5x^2 - 3x^3 + 9.98x^4 + 5.98x^5 + 4x^6
1 + 2x - 5x^2 - 3x^3 + 9.98x^4 + 5.98x^5 + 4x^6 + 9x^7
1 + 2x - 5x^2 - 3x^3 + 9.98x^4 + 5.98x^5 + 4x^6 + 9x^7 + 2x^8
1 + 2x - 5x^2 - 3x^3 + 9.98x^4 + 5.98x^5 + 4x^6 + 9x^7 + 2x^8 - 4x^9
1 + 2x - 5x^2 - 3x^3 + 9.98x^4 + 5.98x^5 + 4x^6 + 9x^7 + 2x^8 - 4x^9 - 6x^10
1 + 2x - 5x^2 - 3x^3 + 9.98x^4 + 5.98x^5 + 4x^6 + 9x^7 + 2x^8 - 4x^9 - 6x^10 - 9x^11
1 + 2x - 5x^2 - 3x^3 + 9.98x^4 + 5.98x^5 + 4x^6 + 9x^7 + 2x^8 - 4x^9 - 6x^10 - 9x^11 - 7x^12
1 + 2x - 5x^2 - 3x^3 + 9.98x^4 + 5.98x^5 + 4x^6 + 9x^7 + 2x^8 - 4x^9 - 6x^10 - 9x^11 - 7x^12 + 9.9x^13
1 + 2x - 5x^2 - 3x^3 + 9.98x^4 + 5.98x^5 + 4x^6 + 9x^7 + 2x^8 - 4x^9 - 6x^10 - 9x^11 - 7x^12 + 9.9x^13 + 8.33x^14
1 + 2x - 5x^2 - 3x^3 + 9.98x^4 + 5.98x^5 + 4x^6 + 9x^7 + 2x^8 - 4x^9 - 6x^10 - 9x^11 - 7x^12 + 9.9x^13 + 8.33x^14 + 16x^15
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$
```

2.3.3 Testfall 3 - evalPoly

romanlum@ubuntu: ~/swo3/UebungMoodle2/aufgabe2

```
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$ ./poly 3
Testfall 3 - evalPoly von  0 - 5
Test-Polynom: 1 + 1x + 3x^2 - 4x^3 + 6.4x^4 + 2x^6 + 6x^7 + 2x^8
Ergebnis bei x = 0: 1
Ergebnis bei x = 1: 17.4
Ergebnis bei x = 2: 1493.4
Ergebnis bei x = 3: 28143.4
Ergebnis bei x = 4: 239003
Ergebnis bei x = 5: 1.28483e+06
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$
```

2.3.4 Testfall 4 - evalPoly - Polynom 2

```
romanlum@ubuntu: ~/swo3/UebungMoodle2/aufgabe2
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$ ./poly 4
Testfall 4 - evalPoly von  0 - 5
Test-Polynom: 1 + 2x - 5x^2 - 3x^3
Ergebnis bei x = 0: 1
Ergebnis bei x = 1: -5
Ergebnis bei x = 2: -39
Ergebnis bei x = 3: -119
Ergebnis bei x = 4: -263
Ergebnis bei x = 5: -489
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$
```

2.3.5 Testfall 5 - polySum - Polynomgrade gleich

```
romanlum@ubuntu: ~/swo3/UebungMoodle2/aufgabe2
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$ ./poly 5
Testfall 5 - polySum von  p+q
Test-Polynom p: 1 + 1x + 3x^2 - 4x^3
Test-Polynom q: 1 + 2x - 5x^2 - 3x^3
Result (3): 2 + 3x - 2x^2 - 7x^3
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$
```

2.3.6 Testfall 6 - polySum - Polynomgrad q > p

```
romanlum@ubuntu: ~/swo3/UebungMoodle2/aufgabe2
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$ ./poly 6
Testfall 6 - polySum von  p+q

Test-Polynom p: 1 + 1x + 3x^2
Test-Polynom q: 1 + 2x - 5x^2 - 3x^3
Result (3): 2 + 3x - 2x^2 - 3x^3
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$
```

2.3.7 Testfall 7 - polySum - Polynomgrad p > q

```
romanlum@ubuntu: ~/swo3/UebungMoodle2/aufgabe2
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$ ./poly 7
Testfall 7 - polySum von  p+q

Test-Polynom p: 1 + 1x + 3x^2 - 4x^3
Test-Polynom q: 1 + 2x - 5x^2
Result (3): 2 + 3x - 2x^2 - 4x^3
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$
```

2.3.8 Testfall 8 - polyProd - Polynomgrade gleich

```
romanlum@ubuntu: ~/swo3/UebungMoodle2/aufgabe2
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$ ./poly 8
Testfall 8 - polyProd von  p und q
Test-Polynom p: 1 + 1x + 3x^2 - 4x^3
Test-Polynom q: 1 + 2x - 5x^2 - 3x^3
Result (6): 1 + 3x - 6x^3 - 26x^4 + 11x^5 + 12x^6
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$
```

2.3.9 Testfall 9 - polyProd - Polynomgrad q > p

```
romanlum@ubuntu: ~/swo3/UebungMoodle2/aufgabe2
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$ ./poly 9
Testfall 9 - polyProd von  p und q

Test-Polynom p: 1 + 1x + 3x^2
Test-Polynom q: 1 + 2x - 5x^2 - 3x^3
Result (5): 1 + 3x - 2x^3 - 18x^4 - 9x^5
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$
```

2.3.10 Testfall 10 - polyProd - Polynomgrad p > q

```
romanlum@ubuntu: ~/swo3/UebungMoodle2/aufgabe2
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$ ./poly 10
Testfall 10 - polyProd von  p und q

Test-Polynom p: 1 + 1x + 3x^2 - 4x^3
Test-Polynom q: 1 + 2x - 5x^2
Result (5): 1 + 3x - 3x^3 - 23x^4 + 20x^5
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$
```

2.3.11 Testfall 11 - polyProdFast - Polynomgrad 3, 7

```
romanlum@ubuntu: ~/swo3/UebungMoodle2/aufgabe2
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$ ./poly 11
Testfall 11 - polyProdFast von  p und q

Test-Polynom p: 1 + 1x + 3x^2 - 4x^3
Test-Polynom q: 1 + 2x - 5x^2 - 3x^3
Result (6): 1 + 3x - 6x^3 - 26x^4 + 11x^5 + 12x^6

Test-Polynom p: 1 + 1x + 3x^2 - 4x^3 + 6.4x^4 + 2x^6 + 6x^7
Test-Polynom q: 1 + 2x - 5x^2 - 3x^3 + 9.98x^4 + 5.98x^5 + 4x^6 + 9x^7
Result (14): 1 + 3x - 6x^3 - 9.62x^4 + 39.76x^5 + 21.92x^6 - 18.18x^7 + 62.952x^8 + 13.272x^9 - 8.44x^10 + 129.44x^11 + 43.88x^12 + 42x^13 + 54x^14
romanlum@ubuntu:~/swo3/UebungMoodle2/aufgabe2$
```