SWO3	Übung zu Softwareentwicklung mit klassischen Sprachen und Bibliotheken 3	WS 2020/21, ÜZ 2

☐ Gruppe M. Hava

☒ Gruppe J. Heinzelreiter   Name: Maximilian Mitter   Aufwand [h]: 10h

☐ Gruppe P. Kulczycki   Peer-Review von: ____________

Beispiel	Lösungsidee (max. 100%)	Implement. (max. 100%)	Testen (max. 100%)
1 (100 P)	100%	100%	80%

Beispiel 1: Rechnen mit Polynomen (`src/poly/`)

Ein Polynom $P(x)$ mit $x \in \mathbb{R}$ vom Grad $m \in \mathbb{N}_0$ hat die allgemeine Form

$$P(x) = p_0 \cdot x^0 + p_1 \cdot x^1 + p_2 \cdot x^2 + \cdots + p_m \cdot x^m = \sum_{i=0}^{m} p_i \cdot x^i$$

wobei $p_i \in \mathbb{R}$ mit $0 \leq i \leq m$ die jeweiligen Koeffizienten der Potenzen von x sind. Ein Polynom vom Grad m kann als C-Array der Länge $m + 1$, welches die Werte der Koeffizienten aufnimmt, dargestellt werden.

Implementieren Sie ein C11-Programm `polynomial`, das die folgenden Funktionalitäten enthält:

a) Schreiben Sie eine Funktion `poly_print`, die ein Polynom auf der Konsole (am Bildschirm) ausgibt. Die Schnittstelle von `poly_print` muss wie folgt aussehen:

```
void poly_print (double const p [], int const m);
```

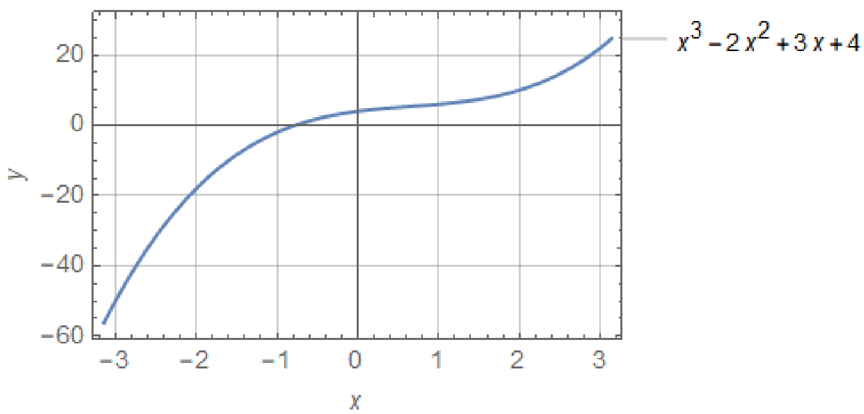
Ein Beispiel: Für das Polynom $P(x) = 4 + 3 \cdot x - 2 \cdot x^2 + x^3$ muss poly_print folgendes ausgeben:

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

b) Schreiben Sie eine Funktion `poly_evaluate`, die ein Polynom an einer gegebenen Stelle $x \in \mathbb{R}$ ausrechnet. Die Schnittstelle von `poly_evaluate` muss wie folgt aussehen:

```
double poly_evaluate (double const p [], int const m, double const x);
```

Ein Beispiel: Für obiges Polynom ergibt sich ein ungefährer Wert von -56.1703, wenn man es an der Stelle $-\pi$ auswertet.



Rechnen mit Polynomen

Allgemeine Hinweise:

Die verwendeten Arrays werden mit der konstanten **ARRAY_SIZE** auf eine fixe Größe von 100 beschränkt.

1. Poly_Print

Lösungsidee:

In einer FOR-Schleife wird das übergebene Array von 0 bis m durchlaufen. Im ersten Durchlauf (i==0) wird nur die Zahl ausgegeben. Beim zweiten Durchgang (i==1) wird „Zahl * x" ausgegeben. Bei den restlichen Durchläufen wird „Zahl * x ^ i" ausgegeben. Innerhalb jeder dieser Möglichkeiten gibt es noch die Überprüfung, ob eine Zahl größer oder kleiner 0 ist. Ist die Zahl genau 0, wird nichts ausgegeben.

Code:

```
void poly_print(double const p[], int const m) {
    printf("P(x) = ");
    for (int i = 0; i < m; i++) {
        if (i == 0) {
            if (p[i] != 0)
                printf("%.2f", p[i]);
        }
        else if (i == 1) {
            if (p[i] > 0)
                printf(" + %.2f*x", p[i]);
            else if (p[i] < 0)
                printf(" - %.2f*x", p[i] * - 1);
        } else {
            if (p[i] > 0)
                printf(" + %.2f*x^%d", p[i], i);
            else if (p[i] < 0)
                printf(" - %.2f*x^%d", p[i] * - 1, i);
        }
    }
    printf("\n");
}
```

Testfälle:

```
Test a1: P(x) = 1.00 + 1.00*x + 3.00*x^2 - 4.00*x^3
Test a2: P(x) = 1.00 + 2.00*x - 5.00*x^2 - 3.00*x^3 + 6.00*x^4 + 3.00*x^5 + 15.00*x^6 + 9115.00*x^7 + 5.00*x^8 - 3654.00*x^9 + 20.00*x^10 + 11.00*x^11
Test a3: P(x) = 1.00 + 1.00*x + 3.00*x^2
Test a4: P(x) =
```

Testfälle mit den folgenden Arrays:

```
double a1[] = {1, 1, 3, -4, 0, 0, 0, 0, 0, 0, 0, 0, 0};
double a2[] = {1, 2, -5, -3, 6, 3, 15, 9115, 5, -3654, 20, 11};
double a3[] = {1, 1, 3, 0, 0, 0};
double a4[] = {0, 0, 0, 0, 0, 0};
```

2. Poly_evaluate

Lösungsidee:

Das übergebene Polynom wird von 0 bis m in einer FOR-Schleife durchlaufen. In jedem durchlauf wird p[i] mit x^i multipliziert und in einer Summenvariable aufsummiert. Diese Variable wird auch zurückgegeben.

Code:

```
double poly_evaluate(double const p[], int const m, double const x) {
    double total = 0;
    for (int i = 0; i < m; i++) {
        total += p[i] * pow(x, i);
    }
    return total;
}
```

Testfälle:

```
Testing with the function P(x) = 1.00 + 1.00*x + 3.00*x^2 - 4.00*x^3
Total at -3.14: 151.49
Total at 3.14: -90.27
Total at 0: 1.00
Total at 10 -3689.00
Total at -10: 4291.00
Total at 250: -62312249.00
```

3. Poly_add

Lösungsidee:

Zu Beginn wird die Größe des größeren Arrays in der Variable **max** gespeichert. Eine FOR-Schleife läuft dann von 0 bis **max** und summiert die übergebenen Polynome an der aktuellen Stelle auf. Annahme: Jedes übergebene Array wird bis zur Stelle **max** mit 0 Initialisiert und kann daher keine ungewünschten Werte liefern.

Code:

```
int poly_add(double const p[], int const m, double const q[], int const n,
             double r[]) {
    int max = m > n ? m : n;
    for (int i = 0; i < max; i++) {
        r[i] = p[i] + q[i];
    }
    return 0;
}
```

Testfälle:

```
Sum of polyadd1 + polyadd2: P(x) = 2.00 + 3.00*x - 2.00*x^2 - 7.00*x^3 - 2.00*x^5
Sum of polyadd2 + polyadd3: P(x) = 1.00 + 2.00*x - 5.00*x^2 - 3.00*x^3 - 2.00*x^5
Sum of polyadd1 + polyadd4: P(x) = 103.00 + 16.00*x + 36.00*x^2 + 748.00*x^3 - 2.00*x^5
Sum of polyadd4 + polyadd3: P(x) = 102.00 + 15.00*x + 33.00*x^2 + 752.00*x^3
```

Für Testfälle verwendete Polynome:

```
double polyadd1[] = {1, 1, 3, -4, 0};
double polyadd2[] = {1, 2, -5, -3, 0, -2};
double polyadd3[] = {0, 0, 0, 0, 0, 0, 0, 0};
double polyadd4[] = {102, 15, 33, 752, 0, 0};
```

Bei jeder Rechnung mit **polyadd3** muss das Ergebnis gleich der anderen Eingabe sein. (Siehe 2. & 4.)

4. Poly_mult

Lösungsidee:

Das Ausgabearray wird bis zur Stelle m + n mit 0 initialisiert. Zwei verschachtelte FOR-Schleifen laufen dann durch die übergebenen Arrays durch und multiplizieren jede Stelle von p mit jeder Stelle in q. Zurückgegeben wird der Grad des Ergebnispolynoms. Dieser ergibt sich aus m + n. Da diese aber die Größe der Arrays angeben und dieser immer 1 mehr ist als der Grad des Polynoms, muss von dieser Zahl noch 2 subtrahiert werden.

Code:

```
int poly_mult(double const p[], int const m, double const q[], int const n,
              double r[]) {
    int i;
    for (i = 0; i < m + n; i++)
        r[i] = 0;

    for (i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
            r[i+j] += p[i] * q[j];
    }
    return m + n - 2;
}
```

Testfälle:

```
Mult of polymult1 * polymult2 (Grade: 8): P(x) = 1.00 + 3.00*x - 6.00*x^3 - 26.00*x^4 + 9.00*x^5 + 10.00*x^6 - 6.00*x^7 + 8.00*x^8
Mult of polymult1 * polymult3 (Grade: 8): P(x) = 1.00 + 1.00*x + 3.00*x^2 - 4.00*x^3
Mult of polymult2 * polymult4 (Grade: 9): P(x) = 102.00 + 219.00*x - 447.00*x^2 + 437.00*x^3 + 1608.00*x^4 - 3435.00*x^5 - 3856.00*x^6 - 1008.00*x^7 - 1504.00*x^8 - 628.00*x^9
Mult of polymult3 * polymult4 (Grade: 9): P(x) = 102.00 + 15.00*x + 33.00*x^2 + 752.00*x^3 + 314.00*x^4
```

Für die Testfälle verwendete Arrays:

```
double polymult1[] = {1, 1, 3, -4, 0, 0};
double polymult2[] = {1, 2, -5, -3, 0, -2};
double polymult3[] = {1, 0, 0, 0, 0, 0, 0};
double polymult4[] = {102, 15, 33, 752, 314};
```

Bei Multiplikationen mit **polymult3** wird das ursprüngliche Array ausgegeben.

5. Poly_mult_fast

Lösungsidee:

Anmerkungen: Diese Funktion funktioniert nur bei *m % 2 == 0*, also Polynomen mit ungeradem Grad. Weiters müssen die Grade der Polynome gleich sein, daher verwendet die Funktion nur **m.** Diese Funktion wurde nach den Erklärungen in der Angabe rekursiv gelöst. Sind die in den Anmerkungen definierten Bedingungen nicht erfüllt, gibt die Funktion sofort -1 zurück. Zu Beginn wird die Hälfte von m bestimmt. Alles von 0 bis **half - 1** wird in die Variablen **Pl** und **Ql** gespeichert. Alles ab **half** bis **m** wird in **Pr** und **Qr** gespeichert. Mit der Funktion poly_mult_fast() werden dann die Hilfsvariablem **Hl** und **Hr** berechnet. Um die Hilfsvariable **Hm** zu berechnen werden zuerst **Pl** und **Pr** addiert und danach mit der Summe von **Ql** und **Qr** multipliziert. Entsprechend der Formel wird dann **Hm** mithilfe der neuen Funktion poly_sub() (Eine Kopie von poly_add() nur mit subtraktion) minus **Hl** und **Hr** gerechnet. Das Ergebnis dieser Rechnung wird dann mit **x^half** multipliziert. Dafür werden die Werte im Array um **half** stellen verschoben. Dieselbe Lösung wird auch für das Multiplizieren von **Hr** mit **x^m** verwendet. Die Ergebnisse dieser beiden Nebenrechnungen werden dann mit **Hl** addiert und der Grad des resultierenden Polynoms zurückgegeben.
Als Abbruchbedingung für die Rekursion wird zu Beginn der Funktion **m == 1** geprüft. Ist diese Bedingung erfüllt, berechnet die Funktion nur das Produkt der beiden übergebenen Funktionen an der Stelle 0.

Code:

```
int poly_mult_fast(double const p[], int const m, double const q[], int const
n, double r[]) {
    if (m != n)
        return -1;
    if (m != 1) {
        if (m % 2 != 0)
            return -1;

        double pl[ARRAY_SIZE], pr[ARRAY_SIZE], ql[ARRAY_SIZE], qr[ARRAY_SIZE],
         hl[ARRAY_SIZE], hr[ARRAY_SIZE], hm[ARRAY_SIZE], hm1[ARRAY_SIZE],
         hm2[ARRAY_SIZE];

        int half = m / 2;

        for (int i = 0; i < half; i++) {
            pl[i] = p[i];
            ql[i] = q[i];
        }
        for (int i = half; i < m; i++) {
            pr[i - half] = p[i];
            qr[i - half] = q[i];
        }

        //Calculate helping variables hl, hr, hm
        poly_mult_fast(pl, half, ql, half, hl);
        poly_mult_fast(pr, m - half, qr, m - half, hr);
        poly_add(pl, half, pr, m - half, hm1);
        poly_add(ql, half, qr, m - half, hm2);
        poly_mult_fast(hm1, m - half, hm2, m - half, hm);
```

```c
        //Calculate result
        double min[ARRAY_SIZE];
        double hrx[ARRAY_SIZE];
        for (int i = 0; i < m * m; i++) {
            min[i] = 0;
            hrx[i] = 0;
        }

        poly_sub(hm, m-half+1, hl, half+1, min);
        poly_sub(min, m-half+1, hr, m-half+1, min);

        int mult = half;
        for (int j = m + mult; j >= mult; j--) {
            min[j] = min[j - mult];
        }
        for (int j = mult - 1; j >= 0; j--) {
            min[j] = 0;
        }

        int mul = m;
        for (int j = m + mul; j >= mul; j--)
            hrx[j] = hr[j - mul];

        for (int j = mul - 1; j >= 0; j--)
            hrx[j] = 0;

        poly_add(hl, half, min, m + mult, r);
        poly_add(r, m + mult, hrx, m + mul, r);
        return m + n - 2;
    } else {
        r[0] = p[0] * q[0];
        return 1;
    }
}
```

Testfälle:

```
Fast Mult of polymultfast1 * polymultfast2 (Grade: 6): P(x) = 1.00 + 3.00*x - 6.00*x^3 - 26.00*x^4 + 11.00*x^5 + 12.00*x^6
Fast Mult of polymultfast1 * polymultfast3 (Grade: 6): P(x) = 1.00 + 1.00*x + 3.00*x^2 - 4.00*x^3 + 12.00*x^6
Fast Mult of polymultfast2 * polymultfast4 (Grade: 6): P(x) = 102.00 + 219.00*x - 447.00*x^2 + 437.00*x^3 + 1294.00*x^4 - 3859.00*x^5 - 2244.00*x^6
Invalid grade of polynome
```

Für die Testfälle verwendete Arrays:

```c
double polymultfast1[] = {1, 1, 3, -4, 0, 0};
double polymultfast2[] = {1, 2, -5, -3, 0, 0};
double polymultfast3[] = {1, 0, 0, 0, 0, 0, 0};
double polymultfast4[] = {102, 15, 33, 752};
```

Beim letzten Testfall wurde polymultfast mit den Größen 1 und 4 aufgerufen. Deshalb wird eine Fehlermeldung ausgegeben und das Programm beendet.