

☐ Gruppe M. Hava☒ Gruppe J. HeinzelreiterName: Neuhold MichaelAufwand [h]: 4☐ Gruppe P. Kulczycki

Beispiel	Lösungsidee (max. 100%)	Implement. (max. 100%)	Testen (max. 100%)
1 (30 P)	100%	100%	100%
2 (5+10+20 P)	100%	95%	95%
3 (35 P)	100%	100%	100%

**Beispiel 1: Hammingfolge (src/hamming/)**

Die Folge der regulären Zahlen  $\langle H_1, H_2, H_3, \dots \rangle$ , in der Informatik *Hammingfolge* genannt (OEIS-Nummer [A051037](#)), ist wie folgt definiert:

1. Es gilt  $H_1 = 1$ .
2. Sei  $H_i, i \in \mathbb{N}$  eine Zahl der Folge. Dann sind auch  $2 \cdot H_i$ ,  $3 \cdot H_i$  und  $5 \cdot H_i$  Zahlen der Folge.

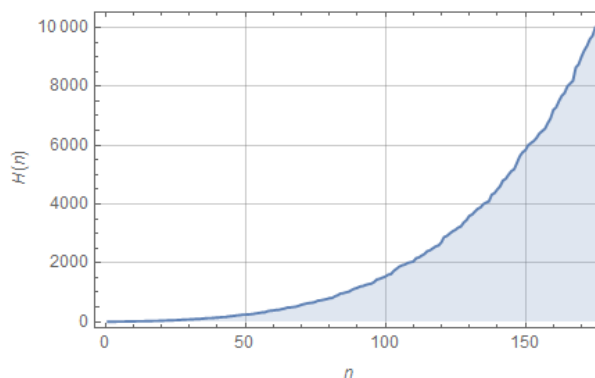
Gesucht ist nun ein möglichst kurzes, einfaches und schnelles C-Programm `hamming_sequence`, welches als Kommandozeilenparameter einen Wert  $Z$  nimmt und die ersten  $n$  Zahlen der Hammingfolge mit  $H_n \leq Z$  aufsteigend sortiert und ohne mehrfaches Vorkommen gleicher Zahlen ausgibt.

*Ein Beispiel:* Der Aufruf von `hamming_sequence` mit  $Z = 30$  liefert die ersten  $n = 18$  Zahlen der Hammingfolge:

1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30

Geben Sie auch die Laufzeit (in Millisekunden) Ihres Algorithmus für verschiedene Werte für  $Z$  an. Verwenden Sie dafür die Funktion `clock` aus der Headerdatei `time.h`.

*Hinweis:* Die Zahlen der Hammingfolge wachsen exponentiell:  $H_n \in \mathcal{O}(b^n)$ ,  $b > 1$ . Es wäre also keine gute Idee, mit einem Feld der Größe  $H_n$  zu arbeiten.



### Beispiel 2: *i*-t größtes Element (src/gross/)

Es ist einfach, das größte (oder kleinste) Element in einem unsortierten Feld (z. B. ganzer Zahlen) mit einem Durchlauf, also in  $\mathcal{O}(n)$ , zu ermitteln. Auch das zweit- (oder dritt-)größte Element kann noch in linearer Zeit einfach ermittelt werden.

*Hinweis:* Sie dürfen im Folgenden davon ausgehen, dass die zu durchsuchenden Felder keine mehrfach vorkommenden Zahlen enthalten.

(a) Implementieren Sie eine C-Funktion

```
int second_largest (int a [], int n);
```

die das zweitgrößte Element in einem unsortierten Feld *a* ganzer Zahlen mit *n* Elementen in einem Durchlauf ermittelt.

(b) Ist man allerdings an dem *i*-t größten Element interessiert, ist es am einfachsten, das Feld absteigend zu sortieren und dann das *i*-te Element herauszugreifen. Implementieren Sie eine C-Funktion

```
int ith_largest_1 (int a [], int n, int i);
```

nach diesem Konzept, wobei Sie zum Sortieren Ihre Funktion `merge_sort` aus Beispiel 3 verwenden müssen.

(c) Der Algorithmus `ith_largest_1` hat eine asymptotische Laufzeitkomplexität von  $\mathcal{O}(n \cdot \log n)$ . Es geht aber auch in linearer Zeit. Erinnern Sie sich zurück an Quick-Sort, der das zu sortierende Feld nach einem Pivotelement in zwei Teilfelder zerlegt (*divide*), beide Teilfelder wieder mittels Quick-Sort sortiert (*conquer*) und damit das gesamte Feld (sogar ganz ohne *combine*) sortiert hat. Implementieren Sie nach diesem Muster eine Funktion

```
int ith_largest_2 (int a [], int n, int i);
```

die zwar mittels Pivotelement eine Zerlegung des Feldes durchführt, dann aber nur jenes Teilfeld weiter betrachtet, in dem das gesuchte Element liegt.

### Beispiel 3: Sortieren ganzer Zahlen (src/sort/)

Bauen Sie den folgenden Quelltext zu einem voll funktionsfähigen C-Programm aus:

```
#define MAX 100

void merge_sort (int a [], int n) {
    // code to sort a[0] .. a[n - 1] using merge sort
}

int main (int argc, char * argv []) {
    int n = 0;
    int a [MAX] = {0};

    // code to read a maximum of MAX values from argv to a and
    // to set n to the actual number of values in a

    // code to display the unsorted array a

    merge_sort (a, n);

    // code to display the sorted array a

    return EXIT_SUCCESS;
}
```

## Inhaltsverzeichnis

<b>1. Hammingfolge .....</b>	<b>2</b>
1.1. Lösungsidee .....	2
1.2. Code .....	2
1.3. Testfälle .....	3
1.3.1. Verschiedene Grenzwerte mit Printf .....	3
1.3.2. Verschiedene Grenzwerte ohne Printf .....	3
1.3.3. Terminalausgabe .....	3
<b>2. i-t größtes Element .....</b>	<b>5</b>
2.1. Lösungsidee .....	5
2.1.1. Lösungsidee second_largest .....	5
2.1.2. Lösungsidee ith_largest_1 .....	5
2.1.3. Lösungsidee ith_largest_2 .....	5
2.2. Code .....	6
2.3. Testfälle .....	11
<b>3. Sortieren ganzer Zahlen .....</b>	<b>12</b>
3.1. Lösungsidee .....	12
3.2. Code .....	12
3.3. Testfälle .....	15
3.3.1. Eingabe weniger als 10 Werte .....	15
3.3.2. Eingabe mehr als 10 Werte .....	15
3.3.3. Eingabe keiner Werte .....	15

## 1. Hammingfolge

### 1.1. Lösungsidee

Das Programm `hamming_sequence` bekommt als Übergabeparameter eine Zahl `Z`. Diese Zahl gibt jene Stelle an bis zu der die Hammingfolge berechnet werden soll. Sollte der Benutzer keinen Grenzwert beim Programmstart mitgeben, so wird eine Fehlermeldung ausgegeben.

Der Algorithmus besteht grundsätzlich aus einer Zählschleife, die bis zur eingegebenen Maximalgrenze alle Zahlen durchgeht. Zu jeder Zahl wird mit einer Funktion („`hamming()`“) ermittelt ob es sich um eine Zahl der Folge handelt oder nicht. Ist sie Teil der Folge, so wird sie direkt in der Konsole ausgegeben. Der Funktion „`hamming`“ wird immer eine Zahl übergeben. In der Funktion wird geprüft, ob sie durch 5, 3 und 2 ohne Rest teilbar ist solange bis sie 1 ist. Sollte dies der Fall sein, so ist die Zahl Teil der Folge und die Funktion liefert `True` zurück.

### 1.2. Code

#### Hamming\_sequence.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// simple standard error function
void PrintErr(char msg[]) {
    printf("<< ERROR (%s) >>\n", msg);
}

// returns true if x is a member of the hamming sequence
int hamming(int x) {
    while(x % 5 == 0)
        x = x / 5;
    while(x % 3 == 0)
        x = x / 3;
    while(x % 2 == 0)
        x = x / 2;
    return x == 1;
}

int main(int argc, char *argv[]) {

    if(argc == 2){
        double start = clock();
        for(int i = 1; i <= atoi(argv[1]); i++){
            if(hamming(i)) { // checking numbers
                printf("%d ", i);
            }
        }
    }
}
```

```

    printf("\ntime: %f ms \n", ((clock() - start) / CLOCKS_PER_SEC) * 1000);
} else {
    PrintErr("Wrong number of parameters!");
}

return 0;
}

```

### 1.3. Testfälle

#### 1.3.1. Verschiedene Grenzwerte mit Printf

10	0.036 ms
100	0.092 ms
1000	0.148 ms
5000	0.529 ms
10000	1.023 ms
1000000	102.522 ms

#### 1.3.2. Verschiedene Grenzwerte ohne Printf

10	0.014 ms
100	0.020 ms
1000	0.096 ms
5000	0.393 ms
10000	0.941 ms
1000000	86.974 ms

#### 1.3.3. Terminalausgabe

Die folgenden Screenshots zeigen einige Terminalausgaben des Programms.

```

hamming — -bash — 95x5
Michaels-MacBook-Pro-3:hamming michaelneuhold$ ./hamming_sequence 100
1 2 3 4 5 6 8 9 10 12 15 16 18 20 24 25 27 30 32 36 40 45 48 50 54 60 64 72 75 80 81 90 96 100
time: 0.069000 ms
[Michaels-MacBook-Pro-3:hamming michaelneuhold$
Michaels-MacBook-Pro-3:hamming michaelneuhold$

```

```
hamming — -bash — 95x5
[Michaels-MacBook-Pro-3:hamming michaelneuhold$ ./hamming_sequence 30 ]
1 2 3 4 5 6 8 9 10 12 15 16 18 20 24 25 27 30
time: 0.034000 ms
Michaels-MacBook-Pro-3:hamming michaelneuhold$
```

```
hamming — -bash — 95x5
[Michaels-MacBook-Pro-3:hamming michaelneuhold$ ./hamming_sequence 10 ]
1 2 3 4 5 6 8 9 10
time: 0.061000 ms
[Michaels-MacBook-Pro-3:hamming michaelneuhold$ ]
Michaels-MacBook-Pro-3:hamming michaelneuhold$
```

```
hamming — -bash — 95x5
[Michaels-MacBook-Pro-3:hamming michaelneuhold$ ./hamming_sequence ]
<< ERROR (Wrong number of parameters!) >>
Michaels-MacBook-Pro-3:hamming michaelneuhold$
```

## 2. i-t größtes Element

### 2.1. Lösungsidee

Für Testzwecke wird eine Feld fixer Größe mit der Funktion `rand()` befüllt. Im Anschluss werden die folgenden Funktionen darauf angewandt:

#### 2.1.1. Lösungsidee `second_largest`

Um das zweitgrößte Element aus einem unsortierten Feld herauszufinden wird lediglich eine Zählschleife benötigt, die das gesamte Feld einmal durchwandert. Zwei Variablen mit `first_largest` and `second_largest` werden mit dem kleinsten Integer Value initialisiert. Wenn in der Schleife dann ein Wert des Arrays betrachtet wird der größer als `first_largest` ist, so wird der Wert von `first_largest` der Variable `second_largest` zugewiesen und `first_largest` bekommt den neuen Wert vom aktuellen Feldelement zugewiesen. Ist jedoch die gerade betrachtete Zahl kleiner als `first_largest` und größer als `second_largest`, so wird lediglich `second_largest` aktualisiert. Ist das Feld fertig durchlaufen, so kann einfach die Variable `second_largest` auf der Konsole ausgegeben werden.

#### 2.1.2. Lösungsidee `ith_largest_1`

Für das Ausgeben des `ith`-größten Elements eines unsortierten Feldes, ist es möglich zunächst das Feld absteigend zu sortieren und anschließend einfach das Element mit dem Index `i-1` auszugeben. Um das Feld zu sortieren wurde (wie in der Angabe gefordert) der Merge-Sort aus Beispiel 3 verwendet. Jedoch wurde statt aufwärts, abwärts sortiert. Dies wurde durch den Tausch des größer-kleiner Operators erzielt ;)

#### 2.1.3. Lösungsidee `ith_largest_2`

Zunächst wird vom Algorithmus geprüft, ob `i = 1` ist. Wenn dem so ist, dann muss das größte Element im Feld zurückgegeben werden. In diesem Fall kann einfach eine lineare Suche durch das Feld verwendet werden. Sollte dies nicht der Fall sein, so kann man das Prinzip des Teilens heranziehen. Dafür wird ein sogenanntes Pivot-Element benötigt. Grundsätzlich gibt es einige Möglichkeiten ein Pivot-Element zu finden. Ich habe mich dafür entschieden, das erste, das letzte und das mittige Element des Feldes zu betrachten und diese im Anschluss in aufsteigender Reihenfolge zu sortieren. Retourniert wird im Anschluss das Mittige. Im Anschluss müssen die restlichen Werte auf der richtigen Seite im Feld „platz nehmen“. Auf diese Weise kann rekursiv vorgegangen werden, bis die Abbruchbedingung `i == 1` erfüllt ist.

## 2.2. Code

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define MAX 10

void printArr(int a[], int n) {
    for(int i = 0; i < n; i++){
        printf("a[%d] = %d ", i, a[i]);
    }
    printf("\n");
    return;
}

void Line() {
    for(int i = 0; i < 50; i++){
        printf("-");
    }
    printf("\n");
    return;
}

void InitArr(int a[], int n) {
    for(int i = 0; i < n; i++){
        a[i] = rand() % 1000;
    }
}

// print error msg
void printErr(char msg[]) {
    printf("<< Error (%s) >>\n", msg);
    return;
}

/* ===== */
// 2.1.

int second_largest(int a[], int n) {
    int f = INT_MIN; // first
    int s = INT_MIN; // second

    if(n >= 2) {
        for(int i = 0; i < n; i++){
            if(a[i] > f){
                s = f;
                f = a[i];
            } else if (a[i] > s){
                s = a[i];
            }
        }
    }
}
```



```

    }
} else {
    printf("ERROR\n");
}
return s;
}

/* ===== */
// 2.2.

void merge(int a[], int from, int mid, int to){
    int tmp[10];
    int k = 0;

    int ai = from;
    int aj = mid+1;

    while(ai <= mid && aj <= to) {
        if(a[ai] >= a[aj]){
            tmp[k++] = a[ai++];
        } else {
            tmp[k++] = a[aj++];
        }
    }

    // copy rest of first subarray
    while(ai <= mid){
        tmp[k++] = a[ai++];
    }

    // copy rest of second subarray
    while(aj <= to){
        tmp[k++] = a[aj++];
    }

    int i,j;
    for(i=from,j=0;i<=to;i++,j++)
        a[i]=tmp[j];
}

void merge_sort_rec(int a[], int from, int to) {
    int mid;
    if (from < to) {
        mid = (from + to) / 2;
        merge_sort_rec(a,from,mid);
        merge_sort_rec(a,mid+1,to);
        merge(a,from,mid,to);
    } else {
        return;
    }
}

```

```

void merge_sort (int a[], int n) {
    merge_sort_rec(a,0,n-1);
    return;
}

int ith_largest_1(int a[], int n, int i) {
    merge_sort(a,n);
    return a[i-1];
}

/* ===== */
// 2.3.

// swap values in array
void swap(int *a, int *b) {
    int h = *a;
    *a = *b;
    *b = h;
}

// return selected pivo element
int pivot_elem(int a[], int left, int right) {
    // pc => pivot candidate
    int pc[3] = {
        a[left],
        a[(left+right)/2],
        a[right]
    };

    // sort candidates
    if (pc[0] > pc[1]) {
        swap(&pc[0], &pc[1]);
    }
    if (pc[1] > pc[2]) {
        swap(&pc[1], &pc[2]);
    }
    if (pc[0] > pc[1]) {
        swap(&pc[0], &pc[1]);
    }

    return pc[1];
}

// brings elements to the correct side of pivot
int split(int a[], int left, int right, int pivot) {
    // split
    do {
        while (a[left] < pivot) {
            left++;
        }
        while (a[right] > pivot) {
            right--;
        }
    } while (left < right);
}

```

```

    }
    swap(&a[left], &a[right]);
} while (left < right);

// return position of pivot
if (a[left] == pivot) return left;
return right;
}

// search largest element in array (linear)
int get_largest(int a[], int left, int right) {
    int max_value = a[left];
    left++;

    for (int i = left; i <= right; i++) {
        if (max_value < a[i]) {
            max_value = a[i];
        }
    }
    return max_value;
}

// search ith-largest element
int search_ith_largest(int a[], int left, int right, int i) {
    if(i == 1) {
        return get_largest(a, left, right);
    }

    // get value of pivot element (candidates: left, middle, right)
    int pivot = pivot_elem(a, left, right);

    // split elements of array --> < / > than pivot and return pivot pos
    int pivot_pos = split(a, left, right, pivot);

    int right_size = right - pivot_pos + 1;

    if (right_size >= i) {
        return search_ith_largest(a, pivot_pos, right, i);
    } else {
        return search_ith_largest(a, left, pivot_pos-1, i-right_size);
    }
}

int ith_largest_2(int a[], int n, int i) {
    return search_ith_largest(a, 0, n-1, i);
}

/* ===== */

int main (int argc, char *argv[]) {

```

```
int a[MAX];

if(argc == 2) {

    int ith = atoi(argv[1]);

    // 1.
    Line();
    InitArr(a,MAX); // init with random numbers
    printf("second_largest => %d\n", second_largest(a,MAX));
    printArr(a,MAX);

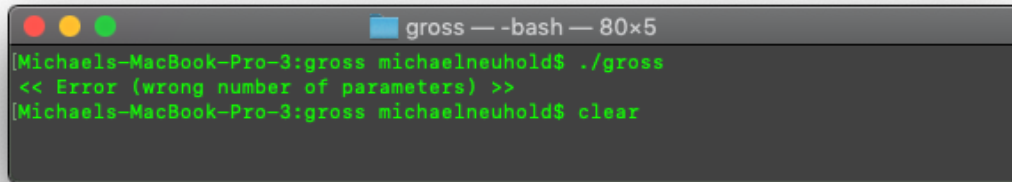
    // 2.
    Line();
    InitArr(a,MAX); // init with random numbers
    printf("ith-largest [i=%d] => %d\n",ith, ith_largest_1(a,MAX,ith));
    printArr(a,MAX);

    // 3.
    Line();
    InitArr(a,MAX); // init with random numbers
    printf("ith-largest [i=%d] => %d\n",ith, ith_largest_2(a,MAX,ith));
    printArr(a,MAX);
} else {
    printErr("wrong number of parameters");
}

return 0;
}
```

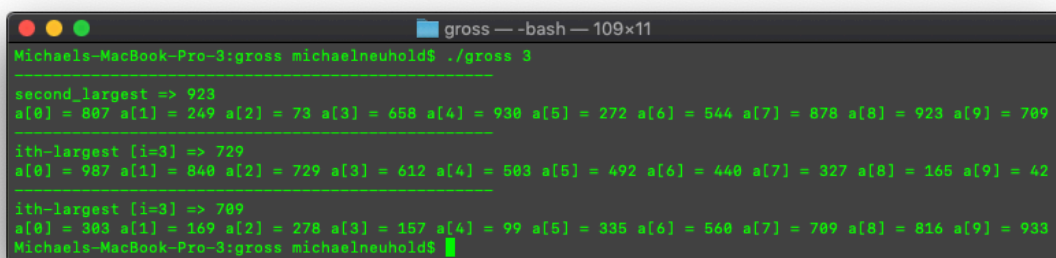
### 2.3. Testfälle

Dieser Testfall zeigt die den Programmaufruf ohne Parameterübergabe. Es ist jedoch ein Parameter notwendig für das ith größte Element. Somit wird eine Fehlermeldung ausgegeben.



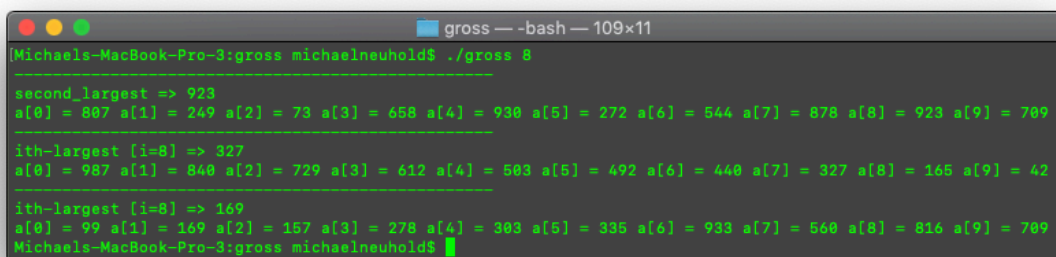
```
gross — -bash — 80x5
[Michaels-MacBook-Pro-3:gross michaelneuhold$ ./gross ]
<< Error (wrong number of parameters) >>
[Michaels-MacBook-Pro-3:gross michaelneuhold$ clear ]
```

Dieser Testfall zeigt eine valide Parameterübergabe. Es wird somit bei den Unterpunkten 2 & 3 jeweils das 3 größte Element der Felder ausgegeben. (Die Felder werden immer zwischen den Funktionsaufrufen mit zufälligen Zahlen initialisiert.)



```
gross — -bash — 109x11
Michaels-MacBook-Pro-3:gross michaelneuhold$ ./gross 3
-----
second_largest => 923
a[0] = 807 a[1] = 249 a[2] = 73 a[3] = 658 a[4] = 930 a[5] = 272 a[6] = 544 a[7] = 878 a[8] = 923 a[9] = 709
-----
ith-largest [i=3] => 729
a[0] = 987 a[1] = 840 a[2] = 729 a[3] = 612 a[4] = 503 a[5] = 492 a[6] = 440 a[7] = 327 a[8] = 165 a[9] = 42
-----
ith-largest [i=3] => 709
a[0] = 303 a[1] = 169 a[2] = 278 a[3] = 157 a[4] = 99 a[5] = 335 a[6] = 560 a[7] = 709 a[8] = 816 a[9] = 933
Michaels-MacBook-Pro-3:gross michaelneuhold$
```

Der folgende Screenshot zeigt nur noch ein weiteres Beispiel mit einem anderen Parameterwert.



```
gross — -bash — 109x11
[Michaels-MacBook-Pro-3:gross michaelneuhold$ ./gross 8 ]
-----
second_largest => 923
a[0] = 807 a[1] = 249 a[2] = 73 a[3] = 658 a[4] = 930 a[5] = 272 a[6] = 544 a[7] = 878 a[8] = 923 a[9] = 709
-----
ith-largest [i=8] => 327
a[0] = 987 a[1] = 840 a[2] = 729 a[3] = 612 a[4] = 503 a[5] = 492 a[6] = 440 a[7] = 327 a[8] = 165 a[9] = 42
-----
ith-largest [i=8] => 169
a[0] = 99 a[1] = 169 a[2] = 157 a[3] = 278 a[4] = 303 a[5] = 335 a[6] = 933 a[7] = 560 a[8] = 816 a[9] = 709
Michaels-MacBook-Pro-3:gross michaelneuhold$
```

### 3. Sortieren ganzer Zahlen

#### 3.1. Lösungsidee

Eingelesen werden die Parameter direkt über die Parameterliste beim Programmaufruf. Es sind lediglich 10 Elemente erlaubt. Wurden mehr als 10 Parameter eingegeben, so wird das Feld mit den ersten 10 Zahlen befüllt.

Um das Feld mit Merge-Sort zu sortieren ist eine Rekursive Funktion notwendig, die jeweils das übergebene Feld in zwei Teile zerlegt. Im Anschluss müssen die Teilfelder natürlich wieder zusammengefügt werden. Das Zusammenfügen basiert prinzipiell auf dem Reißverschlussprinzip. Es werden immer ein Element vom einen Teilfeld mit einem des anderen Teilfeld verglichen. Das kleinere Element wird in ein temporäres Array geschrieben. Der Index des Feldes, von dem das kleinere Element stammt wird im Anschluss um eins erhöht. Dieser Vorgang wird solange wiederholt, bis eines der Teilfelder fertig durchlaufen ist. Im Anschluss müssen noch die restlichen Elemente (falls welche übergeblieben sind) in das temporäre Array geschrieben werden. Zu guter Letzt wird das temporäre Array in das originale Feld (an der richtigen Stelle) zurückgeschrieben.

Die Prototypen (Funktionsdeklarationen) wurden in ein Header-File ausgelagert.

#### 3.2. Code

##### Sort.h

```
#ifndef sort_h
#define sort_h

// prototypes
void line(void);
void printArr(int a[], int n);
void printErr(char msg[]);
void merge(int a[], int from, int mid, int to);
void merge_sort_rec(int a[], int from, int to);
void merge_sort (int a[], int n);

#endif
```

##### Sort.c

```
#include <stdio.h>
#include <stdlib.h>
#include "sort.h"
#define MAX 10
// print error msg
void printErr(char msg[]) {
    printf("<< Error (%s) >>\n", msg);
    return;
}

// Separator
```

```

void line() {
    for(int i = 0; i < 50; i++)
        printf("-");
    printf("\n");
    return;
}

// simply print elements of array separated by spaces
void printArr(int a[], int n) {
    for(int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
    return;
}

// merge two subarrays
void merge(int a[], int from, int mid, int to){
    int tmp[MAX];
    int k = 0;

    int ai = from; // start pos. first subarray
    int aj = mid+1; // start pos. second subarray

    while(ai <= mid && aj <= to) {
        if(a[ai] <= a[aj]){
            tmp[k++] = a[ai++];
        } else {
            tmp[k++] = a[aj++];
        }
    }

    // copy rest of first subarray
    while(ai <= mid){
        tmp[k++] = a[ai++];
    }

    // copy rest of second subarray
    while(aj <= to){
        tmp[k++] = a[aj++];
    }

    // copy tmp into original array
    int i,j;
    for(i=from,j=0;i<=to;i++,j++)
        a[i]=tmp[j];

    return;
}

// mergesort
void merge_sort_rec(int a[], int from, int to) {
    if (from < to) {
        int mid = (from + to) / 2;

```

```

        merge_sort_rec(a,from,mid);
        merge_sort_rec(a,mid+1,to);
        merge(a,from,mid,to);
    }
    return;
}

// call recursive mergesort
void merge_sort (int a[], int n) {
    merge_sort_rec(a,0,n-1);
    return;
}

int main(int argc, char *argv[]) {

    int n;
    int a [MAX] = {0};

    if(argc > 1) {
        // to set n to the actual number of values in a
        if(argc-1 <= 10){
            n = argc-1;
        } else {
            n = 10;
        }

        // code to read a maximum of MAX values from argv to a and
        for(int i = 0; i < n; i++){
            a[i] = atoi(argv[i+1]);
        }

        // code to display the unsorted array a
        line();
        printf("original:\n");
        printArr(a,n);

        // call sorting algorithmus
        merge_sort(a, n);

        // code to display the sorted array a
        line();
        printf("sorted:\n");
        printArr(a,n);

        line();
    } else {
        printf("there is nothing to sort!");
    }
    return EXIT_SUCCESS;
}

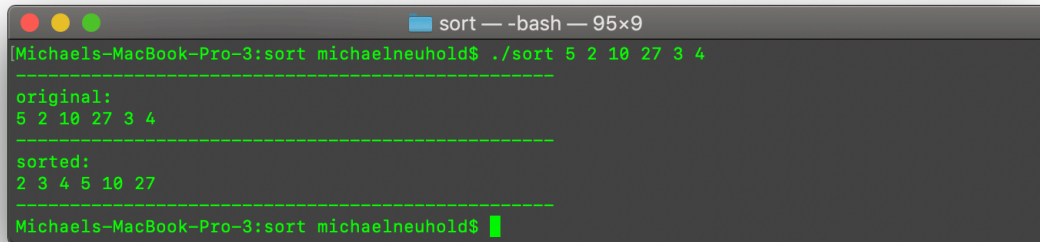
```



### 3.3. Testfälle

#### 3.3.1. Eingabe weniger als 10 Werte

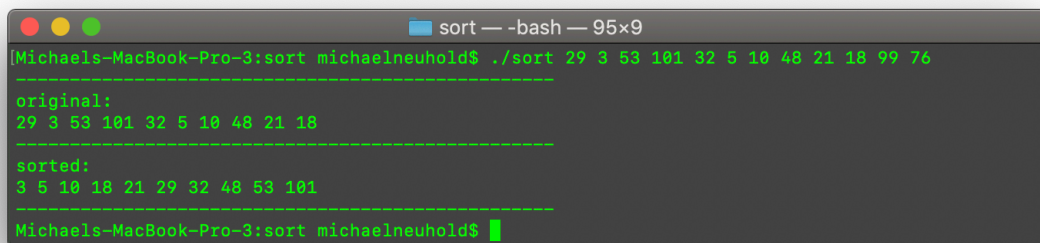
Dieser Testfall zeigt, die Übergabe von ( $n < 10$ ) Parametern. Somit werden alle eingegeben Werte sortiert.



```
sort — -bash — 95x9
[Michaels-MacBook-Pro-3:sort michaelneuhold$ ./sort 5 2 10 27 3 4 ]
-----
original:
5 2 10 27 3 4
-----
sorted:
2 3 4 5 10 27
-----
Michaels-MacBook-Pro-3:sort michaelneuhold$
```

#### 3.3.2. Eingabe mehr als 10 Werte

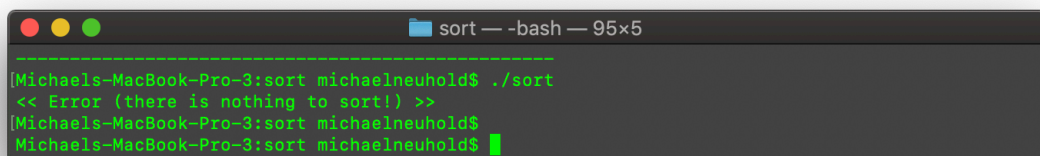
Dieser Testfall zeigt, die Übergabe von ( $n > 10$ ) Parametern. Somit werden lediglich die ersten 10 Werte sortiert.



```
sort — -bash — 95x9
[Michaels-MacBook-Pro-3:sort michaelneuhold$ ./sort 29 3 53 101 32 5 10 48 21 18 99 76 ]
-----
original:
29 3 53 101 32 5 10 48 21 18
-----
sorted:
3 5 10 18 21 29 32 48 53 101
-----
Michaels-MacBook-Pro-3:sort michaelneuhold$
```

#### 3.3.3. Eingabe keiner Werte

Dieser Testfall zeigt einen Programmaufruf ohne Parameter. In diesem Fall wird eine Fehlermeldung ausgegeben.



```
sort — -bash — 95x5
[Michaels-MacBook-Pro-3:sort michaelneuhold$ ./sort ]
<< Error (there is nothing to sort!) >>
[Michaels-MacBook-Pro-3:sort michaelneuhold$ ]
Michaels-MacBook-Pro-3:sort michaelneuhold$
```