

☒ Gruppe 1 (J. Heinzelreiter)☐ Gruppe 2 (M. Hava)Name: PAPESH KonstantinAufwand [h]: 12☐ Gruppe 3 (P. Kulczycki)

Übungsleiter/Tutor: _____

Punkte: _____

Beispiel	Lösungsidee (max. 100%)	Implement. (max. 100%)	Testen (max. 100%)
1 (30 P)	95	100	100
2 (5+10+20 P)	100	100	100
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](https://oeis.org/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.

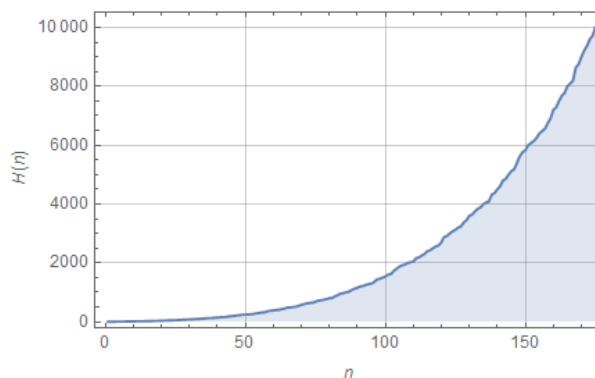
Gesucht ist nun ein möglichst kurzes, einfaches und schnelles C-Programm `hamming_sequence`, welches als Kommandozeilenparameter einen Wert Z nimmt und die ersten n Zahlen der Hammingfolge mit $H_n \leq Z$ aufsteigend sortiert und ohne mehrfaches Vorkommen gleicher Zahlen ausgibt.

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;
}
```

SWO31 & SWB31 Softwareentwicklung mit klassischen Sprachen und Bibliotheken – WS 2018/19

Übungsabgabe 2

Konstantin Papesh

21. Oktober 2018

2.1 Hammingfolge

2.1.1 Lösungsidee

Grundsätzlich wird das Programm als einfaches Konsolenprogramm aufgerufen. Dabei übernimmt es mehrere Parameter, mit welchen es dann arbeitet¹. Diese können mit einfacher Eingabe des Programmes angezeigt werden.

Usage: hamming Z

Weiters überprüft das Programm, ob genug Parameter übergeben wurden. Dann wird überprüft, ob Z größer als 0 ist. Dann wird für die Zeitberechnung die aktuelle Clock gespeichert. Danach wird die Hammingfolge bis zu diesem Z ausgerechnet. Dies geschieht, indem zuerst der erste Index des Hammingarrays auf 1 gesetzt wird. Danach wird bis i größer als Z ist in einer Schleife das aktuelle i mit den Hammingzahlen² multipliziert. Befindet sich i schon in dem Array, wird es nicht hinzugefügt, ansonsten schon. Nachdem die drei Multiplikationen abgeschlossen sind, wird der Index des Hammingarrays erhöht und i auf den nächsten Wert dieses gesetzt³. Ist die Schleife abgeschlossen, wird letztendlich noch die Länge des Hammingarrays mithilfe eines Pointers übergeben. Danach wird das Array sortiert⁴ und erneut die aktuelle Clock gespeichert. Die Endclock wird von der Startclock abgezogen, so ergibt sich das Delta, dieses entspricht der vergangenen Zeit für den Algorithmus. Letztendlich wird diese Zeit und das Array ausgegeben.

¹Siehe auch 2.5

²{2, 3, 5}

³Möglich aufgrund der 2. Regel der Hammingfolge.

⁴Für diesen Algorithmus siehe 2.3

2.1.2 Implementierung

Listing 2.1: main.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <math.h>
5
6 #define MAX 10000
7 #define HAMMING_NUMBERS {2, 3, 5}
8 #define HAMMING_NUMBERS_LENGTH 3
9
10 int inArray(int val, int a[], int length) {
11     for(int i = 0; i < length; i++) {
12         if(a[i] == val)
13             return 1;
14     }
15     return 0;
16 }
17
18 void hamming_sequence(int maxZ, int hammingArr[], int *length) {
19     int n = 1;
20     int i = 1;
21     int h = 0;
22
23     hammingArr[0] = 1;
24     while((i < maxZ) && (n < MAX)) {
25         int multiplierArr[] = HAMMING_NUMBERS;
26         for(int j = 0; j < HAMMING_NUMBERS_LENGTH; j++) {
27             int temp = i * multiplierArr[j];
28             if(temp <= maxZ && !inArray(temp, hammingArr, n)) {
29                 hammingArr[n] = temp;
30                 n++;
31             }
32         }
33         h++;
34         i = hammingArr[h];
35     }
36     *length = n;
37 }
38
39 void merge_sort(int a[], int length) {
40     if(length <= 1)
41         return;
42     else {
43         int left[length];
44         int right[length];
45         int leftMax = ceil((double)length/2);
46         int rightMax = floor((double)length/2);
47
48         for(int i = 0; i < leftMax; i++)
49             left[i] = a[i];
50         for(int i = 0; i < rightMax; i++)
51             right[i] = a[leftMax + i];
52
53         merge_sort(left, leftMax);

```

```

54     merge_sort(right, rightMax);
55
56     int leftIndex = 0;
57     int rightIndex = 0;
58
59     for(int i = 0; i < length; i++) {
60         if(leftIndex >= leftMax) { //left array fully in a[]
61             a[i] = right[rightIndex];
62             rightIndex++;
63         } else if(rightIndex >= rightMax) { //right array fully in a[]
64             a[i] = left[leftIndex];
65             leftIndex++;
66         } else {
67             if(left[leftIndex] < right[rightIndex]) {
68                 a[i] = left[leftIndex];
69                 leftIndex++;
70             } else {
71                 a[i] = right[rightIndex];
72                 rightIndex++;
73             }
74         }
75     }
76 }
77 }
78
79 void printArray(int a[], int n) {
80     for(int i = 0; i < n; i++) {
81         printf("array[%d]=%d\n", i, a[i]);
82     }
83 }
84
85 int main(int argc, char* argv[]) {
86     clock_t start, end;
87     double cpuTime;
88     int maxZ;
89     int hammingArr[MAX];
90     int length;
91
92     start = clock();
93     if(argc != 2) {
94         printf("Usage: %s Z\n", argv[0]);
95         printf("Where Z is the maximum number.\n");
96         return EXIT_FAILURE;
97     }
98     maxZ = atoi(argv[1]);
99     if(maxZ <= 0) {
100         printf("Please enter a number greater than 0!\n");
101         return EXIT_FAILURE;
102     }
103
104     hamming_sequence(maxZ, hammingArr, &length);
105     merge_sort(hammingArr, length);
106     end = clock();
107     printArray(hammingArr, length);
108     cpuTime = ((double) (end-start)) / CLOCKS_PER_SEC;
109     printf("Time taken in seconds: %f\n", cpuTime);
110     return EXIT_SUCCESS;

```

```
111 }
```

2.1.3 Testen

```
khp@KLS:~/git/fh-hgb/ws18/swo/ex02/asgmt/src/hamming$ ./hamming
Usage: ./hamming Z
Where Z is the maximum number.
```

Abbildung 2.1: Fehlermeldung, sollte nur der Programmname angegeben werden.

```
khp@KLS:~/git/fh-hgb/ws18/swo/ex02/asgmt/src/hamming$ ./hamming 30
array[0]=1
array[1]=2
array[2]=3
array[3]=4
array[4]=5
array[5]=6
array[6]=8
array[7]=9
array[8]=10
array[9]=12
array[10]=15
array[11]=16
array[12]=18
array[13]=20
array[14]=24
array[15]=25
array[16]=27
array[17]=30
Time taken in seconds: 0.000017
```

Abbildung 2.2: Ausgabe bei $Z = 30$

```
khp@KLS:~/git/fh-hgb/ws18/swo/ex02/asgmt/src/hamming$ ./hamming 1000000
Time taken in seconds: 0.001378
```

Abbildung 2.3: Rechenzeit bei $Z = 1000000$. Ausgabe des Arrays aufgrund Größe weggelassen.

```
khp@KLS:~/git/fh-hgb/ws18/swo/ex02/asgmt/src/hamming$ ./hamming -2
Please enter a number greater than 0!
```

Abbildung 2.4: Fehlermeldung, sollte Z kleiner gleich 0 sein.

2.2 i-t größtes Element

2.2.1 Lösungsidee

Grundsätzlich wird das Programm als einfaches Konsolenprogramm aufgerufen. Dabei übernimmt es mehrere Parameter, mit welchen es dann arbeitet⁵. Diese können mit einfacher Eingabe des Programmes angezeigt werden.

Usage: sort ith_largest arr1 [aX]

Weiters überprüft das Programm, ob genug Parameter übergeben wurden, oder gar zu viele.⁶

Dann wird das übernommene, noch unbearbeitete Array angezeigt, damit der Nutzer seine Eingabe überprüfen kann.

(a)

Mit der Funktion *second_largest* wird das zweitgrößte Element in dem Array gesucht. Zuerst werden zwei Variablen, *largest* und *secondLargest* deklariert und zugewiesen. Dabei ist nicht der Wert 0 zuzuweisen, sondern *INT_MIN*, da das Array auch aus lediglich negativen Werten bestehen kann. Dann wird das Array einfach von 0..n durchgegangen und der aktuelle Wert jeweils mit den zwei Variablen verglichen. Ist der aktuelle Wert größer als *largest*, wird *largest* mit diesem ersetzt. Ist dies nicht der Fall, wird noch einmal mit *secondLargest* verglichen.

(b)

In diesem Beispiel wird die *merge_sort* Funktion aus 2.3 verwendet. Mit dieser wird das ganze Feld sortiert und dann einfach das gesuchte Element *i* herausgegriffen. Da das Feld schon absteigend sortiert ist, entspricht das *i*-te Element dem Index des Arrays. Jedoch ist beim Funktionsaufruf darauf zu achten, die C Schreibweise⁷ einzuhalten. Also das 1te Element für den Nutzer ist im Code das 0te Element.

(c)

Die Schnittstelle für diese Funktion ist dieselbe wie in 2.2.1, jedoch wird statt des *merge_sort* der *quick_sort* verwendet. Dieser wird selbst implementiert und wurde entsprechend den Anforderungen abgeändert. So wird nicht das ganze Array sortiert, sondern nur so lange, bis man $i - 1$ Elemente über dem Pivotelement hat. Dann entspricht das Pivotelement dem gesuchten Wert. Ist das Array mit den Werten '*Größer als Pivotelement*' größer als das *i*-te Element, bedeutet das, dass das gesuchte *i*-te-Element sich in diesem Array befindet. Daher wird die Funktion rekursiv mit diesem Array aufgerufen. Ansonsten bedeutet es, dass sich das *i*-te Element in dem Array mit den Werten '*Kleiner als Pivotelement*' befindet. Dann wird die Funktion auch wieder rekursiv aufgerufen, jedoch mit dem *smaller*-Array und als gesuchtes Element wird $ithElement - (1 + biggerIndex)$

⁵Siehe auch 2.5

⁶Mit *MAX* wird eine maximale Array-Größe definiert, welche auch überprüft wird.

⁷Beginnt bei 0!

angegeben, da sich ja schon das Array mit den Werten *'Größer als Pivotelement'* + das Pivotelement über dem gesuchten i-th Element befinden.

2.2.2 Implementierung

Listing 2.2: main.c

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <math.h>
4 #include <limits.h>
5
6 #define MAX 5
7 #define ARG_ARRAY_START 2
8
9 void printUsage() {
10     printf("Usage: sort ith_largest arr1 [aX]\n");
11 }
12
13 void printMaxWarning() {
14     printf("Too many values. Array can't be bigger than %d values!\n", MAX);
15 }
16
17 int checkArguments(int argc, char * argv[]) {
18     int success = 0;
19     if(argc < ARG_ARRAY_START + 1) //too few values.
20         printUsage();
21     else if(argc-ARG_ARRAY_START > MAX) //too many values for array.
22         printMaxWarning();
23     else if(atoi(argv[1]) <= 0) { //ith element is smaller than 0. Can't search for that.
24         printf("ith_largest must be bigger than 0!\n");
25         printUsage();
26     }
27     else
28         success = 1;
29     return success;
30 }
31
32 int parseArray(int argc, char* argv[], int a[]) { //parses array. Returns array length or
    0 if error occurs
33     int arrayLength = 0;
34     int i;
35     for(i = ARG_ARRAY_START; i < argc; i++) {
36         a[i-ARG_ARRAY_START] = atoi(argv[i]);
37     }
38     arrayLength = i-ARG_ARRAY_START;
39     return arrayLength;
40 }
41
42 void printArray(int a[], int n) {
43     for(int i = 0; i < n; i++) {
44         printf("array[%d]=%d\n", i, a[i]);
45     }
46 }
47
48 void merge_sort(int a[], int length) {

```



```

49     if(length <= 1)
50         return;
51     else {
52         int left[length];
53         int right[length];
54         int leftMax = ceil((double)length/2);
55         int rightMax = floor((double)length/2);
56
57         for(int i = 0; i < leftMax; i++)
58             left[i] = a[i];
59         for(int i = 0; i < rightMax; i++)
60             right[i] = a[leftMax + i];
61
62         merge_sort(left, leftMax);
63         merge_sort(right, rightMax);
64
65         int leftIndex = 0;
66         int rightIndex = 0;
67
68         for(int i = 0; i < length; i++) {
69             if(leftIndex >= leftMax) { //left array fully in a[]
70                 a[i] = right[rightIndex];
71                 rightIndex++;
72             } else if(rightIndex >= rightMax) { //right array fully in a[]
73                 a[i] = left[leftIndex];
74                 leftIndex++;
75             } else {
76                 if(left[leftIndex] > right[rightIndex]) {
77                     a[i] = left[leftIndex];
78                     leftIndex++;
79                 } else {
80                     a[i] = right[rightIndex];
81                     rightIndex++;
82                 }
83             }
84         }
85     }
86 }
87
88 int second_largest(int a[], int n) {
89     int largest = INT_MIN;
90     int secondLargest = INT_MIN;
91
92     for(int i = 0; i < n; i++) {
93         if(a[i] > largest) {
94             secondLargest = largest;
95             largest = a[i];
96         }
97         else if(a[i] > secondLargest)
98             secondLargest = a[i];
99     }
100     return secondLargest;
101 }
102
103 int ith_largest_1(int a[], int n, int i) {
104     merge_sort(a, n);
105     return a[i];

```

```

106 }
107
108 int ith_largest_2(int a[], int n, int ithElement) {
109     int pivot = a[n-1]; // take last element as pivot
110     int smaller[n];
111     int bigger[n];
112     int smallerIndex = 0;
113     int biggerIndex = 0;
114
115     for(int i = 0; i < n-1; i++) {
116         if(a[i] > pivot) {
117             bigger[biggerIndex] = a[i];
118             biggerIndex++;
119         } else {
120             smaller[smallerIndex] = a[i];
121             smallerIndex++;
122         }
123     }
124
125     if(biggerIndex == ithElement) { //element in bigger array
126         return pivot;
127     } else if(biggerIndex > ithElement) {
128         return ith_largest_2(bigger, biggerIndex, ithElement);
129     } else {
130         return ith_largest_2(smaller, smallerIndex, ithElement - (1 + biggerIndex));
131     }
132 }
133
134 int main(int argc, char * argv[]) {
135     int n = 0;
136     int a[MAX] = {0};
137
138     if(!checkArguments(argc, argv))
139         return EXIT_FAILURE;
140
141     int searchedElement = atoi(argv[1]);
142
143
144     n = parseArray(argc, argv, a);
145
146     printf("### array ###\n");
147     printArray(a, n);
148     printf("\n\n");
149
150
151     printf("2nd largest element=%d\n", second_largest(a, n));
152     printf("%dth largest element=%d\n", searchedElement, ith_largest_1(a, n,
153         searchedElement-1)); // -1 because arrays start at 0. 1st element = arr[0]
153     printf("%dth largest element=%d\n", searchedElement, ith_largest_2(a, n,
154         searchedElement-1)); // -1 because arrays start at 0. 1st element = arr[0]
154     printArray(a, n);
155     return 0;
156 }

```


2.2.3 Testen

```
khp@KLS:~/git/fh-hgb/ws18/swo/ex02/asgmt/src/gross$ ./a.out
Usage: sort ith_largest arr1 [aX]
```

Abbildung 2.5: Fehlermeldung, sollte nur der Programmname angegeben werden.

```
khp@KLS:~/git/fh-hgb/ws18/swo/ex02/asgmt/src/gross$ ./a.out -2
Usage: sort ith_largest arr1 [aX]
```

Abbildung 2.6: Fehlermeldung, sollten zu wenige Parameter angegeben werden.

```
khp@KLS:~/git/fh-hgb/ws18/swo/ex02/asgmt/src/gross$ ./a.out -2 2
ith_largest must be bigger than 0!
```

Abbildung 2.7: Fehlermeldung, sollte das gesuchte Element kleiner 0 sein.

```
khp@KLS:~/git/fh-hgb/ws18/swo/ex02/asgmt/src/gross$ ./a.out 2 20 10 30 25 50
### array ###
array[0]=20
array[1]=10
array[2]=30
array[3]=25
array[4]=50

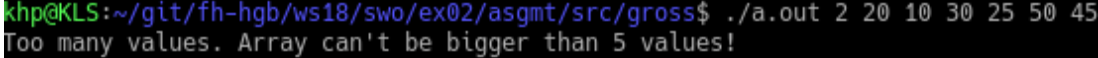
2nd largest element=30
2th largest element=30
2th largest element=30
array[0]=50
array[1]=30
array[2]=25
array[3]=20
array[4]=10
```

Abbildung 2.8: Ausgabe, wenn das 2te Element gesucht wird.

```
khp@KLS:~/git/fh-hgb/ws18/swo/ex02/asgmt/src/gross$ ./a.out 4 20 10 30 25 50
### array ###
array[0]=20
array[1]=10
array[2]=30
array[3]=25
array[4]=50

2nd largest element=30
4th largest element=20
4th largest element=20
array[0]=50
array[1]=30
array[2]=25
array[3]=20
array[4]=10
```

Abbildung 2.9: Ausgabe, wenn das 4te Element gesucht wird.

A terminal window with a black background and green text. The prompt is 'khp@KLS:~/git/fh-hgb/ws18/swo/ex02/asgmt/src/gross\$'. The command entered is './a.out 2 20 10 30 25 50 45'. The output is 'Too many values. Array can't be bigger than 5 values!'.

```
khp@KLS:~/git/fh-hgb/ws18/swo/ex02/asgmt/src/gross$ ./a.out 2 20 10 30 25 50 45
Too many values. Array can't be bigger than 5 values!
```

Abbildung 2.10: Fehlermeldung, sollte das gegebene Array zu groß sein. Größer als Konstante *MAX*, für dieses Beispiel auf 5 gesetzt.

2.3 Sortieren ganzer Zahlen

2.3.1 Lösungsidee

Grundsätzlich wird das Programm als einfaches Konsolenprogramm aufgerufen. Dabei übernimmt es mehrere Parameter, mit welchen es dann arbeitet⁸. Diese können mit einfacher Eingabe des Programmes angezeigt werden.

Usage: sort a1 [aX]

Weiters überprüft das Programm, ob genug Parameter übergeben wurden, oder gar zu viele.⁹

Dann wird das übernommene, noch unbearbeitete Array angezeigt, damit der Nutzer seine Eingabe überprüfen kann.

Dieses wird darauffolgend sortiert, und erneut ausgegeben.

Da in C Arrays als Pointer übergeben werden, muss (oder sogar kann) garnicht mit *return* gearbeitet werden. Somit wird die Funktion *merge_sort* als void aufgerufen. In dieser wird rekursiv das Array geteilt, bis nur noch zwei Elemente vorhanden sind. Diese werden dann verglichen und sortiert. Dann wird aufgestiegen und wieder die beiden Arrays verglichen. Dabei werden jeweils zwei Elemente der beiden Arrays verglichen. Das kleinere Element wird dann an ein weiteres Array, *a[]*, gehängt und der Index des Arrays, welches dieses Element enthalten hat, erhöht. Dies geht so lange, bis der Index einer der beiden Arrays die Länge des jeweiligen Arrays erreicht hat. Dies bedeutet, dass alle Elemente dieses Arrays sich bereits in *a[]* befinden. Daraus kann geschlossen werden, dass alle restlichen Elemente im zweiten Array größer sind als alle im *a[]* enthaltenen. Also können alle Elemente des zweiten Arrays in Reihenfolge an das Array *a[]* angehängt werden. Dies wird rekursiv so lange wiederholt, bis das Originalarray sortiert ist.

2.3.2 Implementierung

Listing 2.3: main.c

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <math.h>
4
5 #define MAX 100
6
7 void printUsage() {
8     printf("Usage: sort a1 [aX]\n");
9 }
10
11 void printMaxWarning() {
12     printf("Too many values. Array can't be bigger than %d values!\n", MAX);
13 }
14
15 int checkArguments(int argc) {
16     int success = 0;
17     if(argc < 2)
```

⁸Siehe auch 2.11

⁹Mit *MAX* wird eine maximale Array-Größe definiert, welche auch überprüft wird.

```

18     printUsage();
19     else if(argc-1 > MAX)
20         printMaxWarning();
21     else
22         success = 1;
23     return success;
24 }
25
26 int parseArray(int argc, char* argv[], int a[]) { //parses array. Returns array length or
    0 if error occurs
27     int arrayLength = 0;
28     int i;
29     for(i = 1; i < argc; i++) {
30         a[i-1] = atoi(argv[i]);
31     }
32     arrayLength = i-1;
33     return arrayLength;
34 }
35
36 void printArray(int a[], int n) {
37     for(int i = 0; i < n; i++) {
38         printf("array[%d]=%d\n", i, a[i]);
39     }
40 }
41
42 void merge_sort(int a[], int length) {
43     if(length <= 1)
44         return;
45     else {
46         int left[length];
47         int right[length];
48         int leftMax = ceil((double)length/2);
49         int rightMax = floor((double)length/2);
50
51         for(int i = 0; i < leftMax; i++)
52             left[i] = a[i];
53         for(int i = 0; i < rightMax; i++)
54             right[i] = a[leftMax + i];
55
56         merge_sort(left, leftMax);
57         merge_sort(right, rightMax);
58
59         int leftIndex = 0;
60         int rightIndex = 0;
61
62         for(int i = 0; i < length; i++) {
63             if(leftIndex >= leftMax) { //left array fully in a[]
64                 a[i] = right[rightIndex];
65                 rightIndex++;
66             } else if(rightIndex >= rightMax) { //right array fully in a[]
67                 a[i] = left[leftIndex];
68                 leftIndex++;
69             } else {
70                 if(left[leftIndex] > right[rightIndex]) {
71                     a[i] = left[leftIndex];
72                     leftIndex++;
73                 } else {

```

```
74             a[i] = right[rightIndex];
75             rightIndex++;
76         }
77     }
78 }
79 }
80 }
81
82 int main(int argc, char * argv[]) {
83     int n = 0;
84     int a[MAX] = {0};
85
86     if(!checkArguments(argc))
87         return EXIT_FAILURE;
88
89     n = parseArray(argc, argv, a);
90
91     printf("### Unsorted array ###\n");
92     printArray(a, n);
93     printf("\n\n");
94
95     merge_sort(a, n);
96
97     printf("### Sorted array ###\n");
98     printArray(a, n);
99     printf("\n\n");
100
101     return EXIT_SUCCESS;
102 }
```


2.3.3 Testen

```
khp@KLS:~/git/fh-hgb/ws18/swo/ex02/asgmt/src/sort$ ./sort
Usage: sort a1 [aX]
```

Abbildung 2.11: Fehlermeldung, sollte nur der Programmname angegeben werden.

```
khp@KLS:~/git/fh-hgb/ws18/swo/ex02/asgmt/src/sort$ ./sort 1 -4 43 -543 -53 31
### Unsorted array ###
array[0]=1
array[1]=-4
array[2]=43
array[3]=-543
array[4]=-53
array[5]=31

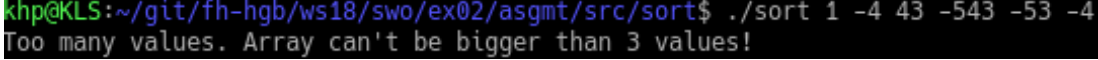
### Sorted array ###
array[0]=43
array[1]=31
array[2]=1
array[3]=-4
array[4]=-53
array[5]=-543
```

Abbildung 2.12: Sortiertes Array mit negativen Werten.

```
khp@KLS:~/git/fh-hgb/ws18/swo/ex02/asgmt/src/sort$ ./sort 1 -4 43 -543 -53 -4
### Unsorted array ###
array[0]=1
array[1]=-4
array[2]=43
array[3]=-543
array[4]=-53
array[5]=-4

### Sorted array ###
array[0]=43
array[1]=1
array[2]=-4
array[3]=-4
array[4]=-53
array[5]=-543
```

Abbildung 2.13: Sortiertes Array mit mehreren gleichen Werten



```
khp@KLS:~/git/fh-hgb/ws18/swo/ex02/asgmt/src/sort$ ./sort 1 -4 43 -543 -53 -4
Too many values. Array can't be bigger than 3 values!
```

Abbildung 2.14: Fehlermeldung, sollte das gegebene Array zu groß sein. Größer als Konstante *MAX*, für dieses Beispiel auf 5 gesetzt.