Ausarbeitung Übung 02

# Hammingfolge

## Lösungsidee

Alle Zahlen der hamming Folge lassen sich als Ergebnis des Terms berechnen. Man kann also drei verschachtelte while-Schleifen anschreiben, welche jeweils einen anderen Exponenten (x, y oder z) erhöhen. Abbruchbedingung ist dann das Erreichen des Wertes 10. Zusätzlich wird in der innersten Schleife abgebrochen, wenn der neu errechnete Wert den an das Programm als Argument übergebenen Maximalwert übersteigt.   
Jeder errechnete Wert wird in ein statisches Feld der Größe 1000 eingetragen, da bis zum Maximalwert eines 4 Byte Integers nie eine Sequenz von über 1000 Hamming-Zahlen entsteht.  
Durch die Kombination von Schleifen werden auch Duplikate vermieden, weil nie eine Permutation des Tupels (x, y, z) zur selben Zahl führt.  
Zuletzt wird das Feldmit Hilfe der Funktion *qsort* noch sortiert und anschließend ausgegeben.

## Implementierung (hamming.c)

#include **<stdio.h>**#include **<stdlib.h>**#include **<math.h>**#include **<time.h>**#define **HAMMING\_SEQ\_ARRAY\_LENGTH** 1000  
  
*/\*\*  
 \* @param exp\_2 The exponent for the component 2.  
 \* @param exp\_3 The exponent for the component 3.  
 \* @param exp\_5 The exponent for the component 5.  
 \* @return The hamming number according to the exponents  
 \* following the forumla 2<sup>exp\_2</sup> \*  
 \* 3<sup>exp\_3</sup> \* 5<sup>exp\_5</sup>.  
 \*/***unsigned int** calc\_hamming(**int** exp\_2, **int** exp\_3, **int** exp\_5)  
{  
 **return** (**unsigned int**) (pow(2, exp\_2) \* pow(3, exp\_3) \* pow(5, exp\_5));  
}  
  
*/\*\*  
 \* @param p The void pointer to cast.  
 \* @return The int value pointed to by <i>p</i>.  
 \*/***int** cast\_int(**const void** \***const** p)  
{  
 **return** \*(**int**\*)p;  
}  
  
*/\*\*  
 \* A binary comparison "predicate" for integer numbers.  
 \*/***int** compare\_int(**const void** \*a, **const void** \*b)  
{  
 **if** (cast\_int(a) < cast\_int(b)) {  
 **return** -1;  
 } **else if** (cast\_int(a) > cast\_int(b)) {  
 **return** 1;  
 } **else** {  
 **return** 0;  
 }  
}  
  
**int** main(**int** argc, **char** \*argv[])  
{  
 clock\_t t;  
 t = clock();  
  
 **if** (argc != 2) {  
 printf(**"Invalid number of arguments.\n"**);  
 **return EXIT\_FAILURE**;  
 }  
  
 **unsigned int** z = (**unsigned int**) atoi(argv[1]); *// NOLINT(cert-err34-c)* **if** (z <= 0 || z > **INT\_MAX**) {  
 printf(**"The upper bound must be an integer number x where 0 < x < %d."**, **INT\_MAX**);  
 **return EXIT\_FAILURE**;  
 }  
  
 *// The array is large enough to  
 // hold all hamming numbers up to  
 // INT\_MAX* **unsigned int** hamming\_numbers[**HAMMING\_SEQ\_ARRAY\_LENGTH**];  
 size\_t last = 0;  
 **int** exp\_2 = 0, exp\_3 = 0, exp\_5 = 0;  
 **unsigned int** current = 0;  
  
 *// calculate all numbers in the sequence  
 // and add them to the values array* **while** (exp\_2 < 10) {  
 exp\_3 = 0;  
 **while** (exp\_3 < 10) {  
 exp\_5 = 0;  
 current = 0;  
 *// <last> will always be less than HAMMING\_SEQ.., just making sure  
 // not to segfault in case I am in error.* **while** (current < z && exp\_5 < 10 && last < **HAMMING\_SEQ\_ARRAY\_LENGTH**) {  
 current = calc\_hamming(exp\_2, exp\_3, exp\_5++);  
 **if** (current < z) {  
 hamming\_numbers[last++] = current;  
 }  
 }  
 ++exp\_3;  
 }  
 ++exp\_2;  
 }  
  
 qsort(hamming\_numbers, last, **sizeof**(**int**), compare\_int);  
  
 **for** (size\_t i = 0; i < last; ++i) {  
 printf(**"%d"**, hamming\_numbers[i]);  
 **if** (i + 1 < last) {  
 printf(**", "**);  
 }  
 }  
 printf(**"\n"**);  
  
 t = clock() - t;  
 printf(**"Time passed: %.2fms"**, ((**float**) t) / **CLOCKS\_PER\_SEC** \* 1000);  
  
 **return EXIT\_SUCCESS**;  
}

## Tests

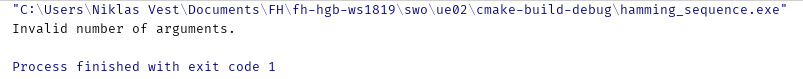


Figure 1 Aufruf ohne Argument

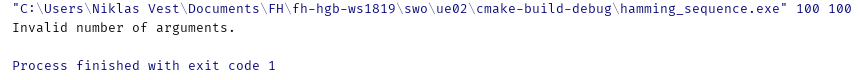
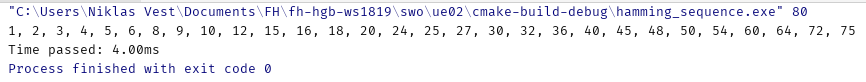
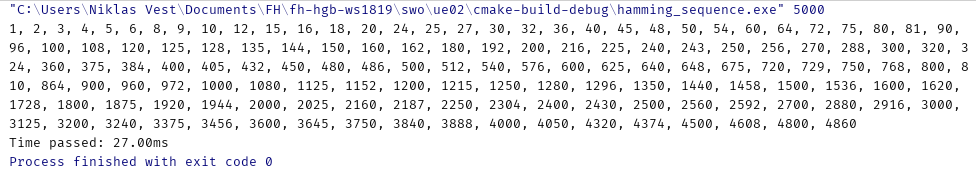


Figure 2 Aufruf mit zu vielen Argumenten



Figure 3 Aufruf mit invalidem Argument





# *i*-t größtes Element

## Lösungsidee

1. Das Parameter-Feld *a* wird sequentiell durchsucht. Die Hilfsvariablen *max* und *second\_to\_max* werden mit dem kleinst-möglichen Integer Wert initialisiert. Wird ein Wert gefunden der größer als *second\_to\_max* ist, kann man davon ausgehen, dass der Wert der zweitgrößte oder sogar der größte Wert ist. Aus diesem Grund wird der bis jetzt bekannte Maximalwert noch geprüft. Ist der neue Wert größer, ist er der neue Maximalwert und der alte wird zum zweitgrößten Wert. Ist er ohnehin nur der zweitgrößte wird er direkt der variable *second\_to\_max* zugewiesen. Der Rückgabewert ergibt sich aus *second\_to\_max*.
2. Man sortiert das übergebene Feld mittels Merge-Sort aus Aufgabe 3 und indiziert dann „gespiegelt“ mit *i*. Da die Merge-Sort Funktion aufsteigend sortiert, muss das i-te Element von hinten zurückgegeben werden. Anmerkung: Der algorithmus wurde zumindest konzeptionell von [Tutorialspoint](https://www.tutorialspoint.com/data_structures_algorithms/merge_sort_algorithm.htm) bezogen. Es hätte auf der Seite zwar eine C-Beispielimplementierung gegeben, dann wäre aber garkeine Übung dabei gewesen.
3. Code für den Quickselect-Algorithmus wurde wieder konzeptionell aus dem Internet bezogen, diesmal von [Wikipedia](https://en.wikipedia.org/wiki/Quickselect). Dadurch, dass der Quickselect-Algorithmus auch das i-t kleinste Element auswählt, muss der Index wieder gespiegelt werden (siehe Implementierung).

## Implementierung (gross.c)

#include **<stdio.h>**#include **<stdlib.h>**#include **<math.h>***// region merge sort***void** merge(**int** \***const** a, **int** a\_n, **const int** \***const** b, **int** b\_n)  
{  
 **int** \*merged = (**int** \*) malloc(**sizeof**(**int**) \* (size\_t) (a\_n + b\_n));  
 size\_t merged\_ind = 0;  
 **int** i = 0;  
 **int** j = 0;  
 **while** (i < a\_n && j < b\_n) {  
 **if** (a[i] > b[j]) {  
 merged[merged\_ind] = b[j++];  
 } **else** {  
 merged[merged\_ind] = a[i++];  
 }  
 ++merged\_ind;  
 }  
 **while** (i < a\_n) {  
 merged[merged\_ind++] = a[i++];  
 }  
 **while** (j < b\_n) {  
 merged[merged\_ind++] = b[j++];  
 }  
 **for** (size\_t override = 0; override < merged\_ind; ++override) {  
 a[override] = merged[override];  
 }  
 free(merged);  
}  
  
**void** merge\_sort(**int** a[], **int** n)  
{  
 **if** (n != 1) {  
 **int** \***const** a\_left = a;  
 **int** left = n / 2;  
  
 **int** \***const** a\_right = a + (n / 2);  
 **int** right = n - left;  
  
 merge\_sort(a\_left, left);  
 merge\_sort(a\_right, right);  
  
 merge(a\_left, left, a\_right, right);  
 }  
}  
  
*// endregion  
  
// region quickselect***int** partition(**int** \***const** a, **int** left, **int** right, **int** pivot\_index)  
{  
 **int** pivot\_value = a[pivot\_index];  
 **int** buff = a[pivot\_index];  
 a[pivot\_index] = a[right];  
 a[right] = buff;  
 **int** store\_index = left;  
 **for** (**int** i = left; i < right; ++i) {  
 **if** (a[i] < pivot\_value) {  
 buff = a[store\_index];  
 a[store\_index] = a[i];  
 a[i] = buff;  
 ++store\_index;  
 }  
 }  
 buff = a[right];  
 a[right] = a[store\_index];  
 a[store\_index] = buff;  
 **return** store\_index;  
}  
  
**int** select(**int** \***const** a, **int** left, **int** right, **int** k)  
{  
 **if** (left == right) {  
 **return** a[left];  
 }  
 **int** pivot\_index = left + (**int**) floor(rand() % (right - left + 1));  
 pivot\_index = partition(a, left, right, pivot\_index);  
 **if** (k == pivot\_index) {  
 **return** a[k];  
 } **else if** (k < pivot\_index) {  
 **return** select(a, left, pivot\_index - 1, k);  
 } **else** {  
 **return** select(a, pivot\_index + 1, right, k);  
 }  
}  
  
*// endregion  
  
// region exercises  
  
/\*\*  
 \* Finds the second largest number using  
 \* linear search.  
 \*/***int** second\_largest(**const int** \***const** a , **int** n)  
{  
 **int** max = **INT\_MIN**;  
 **int** second\_to\_max = **INT\_MIN**;  
 **for** (**int** i = 0; i < n; ++i) {  
 **int** current = a[i];  
 **if** (current > second\_to\_max) {  
 **if** (current > max) {  
 second\_to\_max = max;  
 max = current;  
 } **else** {  
 second\_to\_max = current;  
 }  
 }  
 }  
 **return** second\_to\_max;  
}  
  
*/\*\*  
 \* Finds the <i>i</i>th-largest number  
 \* by sorting the array using the merge  
 \* sort alrorithm first, then indexes  
 \* using <i>i</i>.  
 \*/***int** ith\_largest\_1(**int** \***const** a, **int** n, **int** i)  
{  
 merge\_sort(a, n);  
 **return** a[n - i];  
}  
  
*/\*\*  
 \* Finds the <i>i</i>th-largest number  
 \* using the quickselect algorithm.  
 \*/***int** ith\_largest\_2(**int** \***const** a, **int** n, **int** i)  
{  
 **return** select(a, 0, n-1, n - i);  
}  
  
*// endregion***int** main()  
{  
 **int** arr[10] = {98, 4, 3, 80, 2, 9, -3, 10, 6, 99};  
 printf(**"Second largest: %d\n"**, second\_largest(arr, 10));  
 printf(**"4th largest: %d\n"**, ith\_largest\_1(arr, 10, 4));  
 printf(**"3rd largest: %d\n"**, ith\_largest\_2(arr, 10, 3));  
}

## Tests

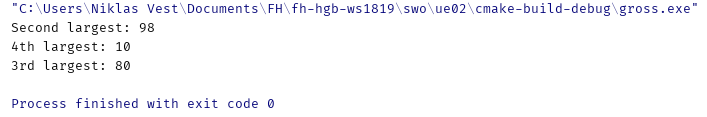


Figure 4 Testfälle aus der main Funktion

# Sortieren ganzer Zahlen

## Lösungsidee

Nach dem Motto „Besser schlau als hart arbeiten“ habe ich wie bei Aufgabe 2 bereits angemerkt die algorithmische Vorgehensweise von [Wikipedia](https://en.wikipedia.org/wiki/Merge_sort).

## Implementierung (sort.c)

#include **<stdlib.h>**#include **<stdio.h>**#define **MAX** 100  
  
**void** merge(**int** \***const** a, **int** a\_n, **int** \***const** b, **int** b\_n)  
{  
 **int** \*merged = (**int** \*) malloc(**sizeof**(**int**) \* (size\_t) (a\_n + b\_n));  
 size\_t merged\_ind = 0;  
 **int** i = 0;  
 **int** j = 0;  
 **while** (i < a\_n && j < b\_n) {  
 **if** (a[i] > b[j]) {  
 merged[merged\_ind] = b[j++];  
 } **else** {  
 merged[merged\_ind] = a[i++];  
 }  
 ++merged\_ind;  
 }  
  
 **while** (i < a\_n) {  
 merged[merged\_ind++] = a[i++];  
 }  
  
 **while** (j < b\_n) {  
 merged[merged\_ind++] = b[j++];  
 }  
  
 **for** (size\_t override = 0; override < merged\_ind; ++override) {  
 a[override] = merged[override];  
 }  
  
 free(merged);  
}  
  
**void** merge\_sort(**int** a[], **int** n)  
{  
 **if** (n != 1) {  
 **int** \***const** a\_left = a;  
 **int** left = n / 2;  
  
 **int** \***const** a\_right = a + (n / 2);  
 **int** right = n - left;  
  
 merge\_sort(a\_left, left);  
 merge\_sort(a\_right, right);  
  
 merge(a\_left, left, a\_right, right);  
 }  
}  
  
**int** main(**int** argc, **char** \*argv[])  
{  
 **int** n = 0;  
 **int** a[**MAX**] = {0};  
  
 **if** (argc < 2) {  
 printf(**"What am I supposed to sort?\n"**);  
 **return EXIT\_FAILURE**;  
 }  
  
 **if** (argc > **MAX**) {  
 printf(**"Cannot sort so many numbers. Enter a max of %d values.\n"**, **MAX**);  
 **return EXIT\_FAILURE**;  
 }  
  
 **int** success = 1;  
 **while** (n < argc - 1 && n < **MAX** && success) {  
 success = sscanf(argv[n + 1], **"%d"**, a + n); *// NOLINT(cert-err34-c)* ++n;  
 }  
  
 **if** (!success) {  
 printf(**"\"%s\" is not an integer number.\n"**, argv[n]);  
 **return EXIT\_FAILURE**;  
 }  
  
 printf(**"Unsorted: "**);  
 **for** (**int** i = 0; i < n; ++i) {  
 printf(**"%d"**, a[i]);  
 **if** (i + 1 < n) {  
 printf(**", "**);  
 }  
 }  
 printf(**"\n"**);  
  
 merge\_sort(a, n);  
  
 printf(**"Sorted: "**);  
 **for** (**int** i = 0; i < n; ++i) {  
 printf(**"%d"**, a[i]);  
 **if** (i + 1 < n) {  
 printf(**", "**);  
 }  
 }  
  
 **return EXIT\_SUCCESS**;  
}

## Tests

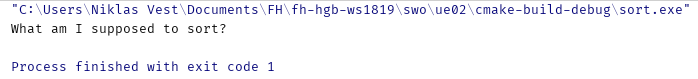


Figure 5 Aufruf mit zu wenigen Argumenten

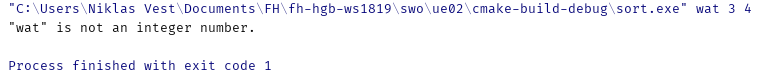


Figure 6 Aufruf mit invaliden Argumenten

