Ausarbeitung Übung 02

Hammingfolge

Lösungsidee

Alle Zahlen der hamming Folge lassen sich als Ergebnis des Terms $2^x \times 3^y \times 5^z$ 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)

```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

```
"C:\Users\Niklas Vest\Documents\FH\fh-hgb-ws1819\swo\ue02\cmake-build-debug\hamming_sequence.exe"
Invalid number of arguments.

Process finished with exit code 1
```

Figure 1 Aufruf ohne Argument

```
"C:\Users\Niklas Vest\Documents\FH\fh-hgb-ws1819\swo\ue02\cmake-build-debug\hamming_sequence.exe" 100 100
Invalid number of arguments.

Process finished with exit code 1
```

Figure 2 Aufruf mit zu vielen Argumenten

```
"C:\Users\Niklas Vest\Documents\FH\fh-hgb-ws1819\swo\ue02\cmake-build-debug\hamming_sequence.exe" wat
The upper bound must be an integer number x where 0 < x < 2147483647.
Process finished with exit code 1
```

Figure 3 Aufruf mit invalidem Argument

```
"C:\Users\Niklas Vest\Documents\FH\fh-hgb-ws1819\swo\ue02\cmake-build-debug\hamming_sequence.exe" 80
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
Time passed: 4.00ms
Process finished with exit code 0
```

```
"C:\Users\Niklas Vest\Documents\FH\fh-hgb-ws1819\swo\ue02\cmake-build-debug\hamming_sequence.exe" 5000
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, 108, 120, 125, 128, 135, 144, 150, 160, 162, 180, 192, 200, 216, 225, 240, 243, 250, 256, 270, 288, 300, 320, 3
24, 360, 375, 384, 400, 405, 432, 450, 480, 486, 500, 512, 540, 576, 600, 625, 640, 648, 675, 720, 729, 750, 768, 800, 8
10, 864, 900, 960, 972, 1000, 1080, 1125, 1152, 1200, 1215, 1250, 1280, 1296, 1350, 1440, 1458, 1500, 1536, 1600, 1620,
1728, 1800, 1875, 1920, 1944, 2000, 2025, 2160, 2187, 2250, 2304, 2400, 2430, 2500, 2560, 2592, 2700, 2880, 2916, 3000,
3125, 3200, 3240, 3375, 3456, 3600, 3645, 3750, 3840, 3888, 4000, 4050, 4320, 4374, 4500, 4608, 4800, 4860
Time passed: 27.00ms
Process finished with exit code 0
```

i-t größtes Element

Lösungsidee

a) 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*.

b) 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 bezogen. Es hätte auf der Seite zwar eine C-Beispielimplementierung gegeben, dann wäre aber garkeine Übung dabei gewesen.

c) Code für den Quickselect-Algorithmus wurde wieder konzeptionell aus dem Internet bezogen, diesmal von Wikipedia. Dadurch, dass der Quickselect-Algorithmus auch das i-t kleinste Element auswählt, muss der Index wieder gespiegelt werden (siehe Implementierung).

Implementierung (gross.c)

```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

```
"C:\Users\Niklas Vest\Documents\FH\fh-hgb-ws1819\swo\ue02\cmake-build-debug\gross.exe"
Second largest: 98
4th largest: 10
3rd largest: 80

Process finished with exit code 0
```

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.

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

```
"C:\Users\Niklas Vest\Documents\FH\fh-hgb-ws1819\swo\ue02\cmake-build-debug\sort.exe"
What am I supposed to sort?

Process finished with exit code 1
```

Figure 5 Aufruf mit zu wenigen Argumenten

```
"C:\Users\Niklas Vest\Documents\FH\fh-hgb-ws1819\swo\ue02\cmake-build-debug\sort.exe" wat 3 4
"wat" is not an integer number.

Process finished with exit code 1
```

Figure 6 Aufruf mit invaliden Argumenten

```
"C:\Users\Niklas Vest\Documents\FH\fh-hgb-ws1819\swo\ue02\cmake-build-debug\sort.exe" 0 30 22 4 -10 8 3 70 99 5
Unsorted: 0, 30, 22, 4, -10, 8, 3, 70, 99, 5
Sorted: -10, 0, 3, 4, 5, 8, 22, 30, 70, 99
Process finished with exit code 0
```

```
"C:\Users\Niklas Vest\Documents\FH\fh-hgb-ws1819\swo\ue02\cmake-build-debug\sort.exe" 100 3 -5 83 30 1 -55 39 25 111
Unsorted: 100, 3, -5, 83, 30, 1, -55, 39, 25, 111
Sorted: -55, -5, 1, 3, 25, 30, 39, 83, 100, 111
Process finished with exit code 0
```