```c
#include "util.h"
#include "vector.h"

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

/**
 * @brief Sorts a given array with N elements using radixsort with the LSD method
 * @note 1.1 a) Sorting happens in these steps:
 * 1. For every digit 0 to 9, create a dynamic array for the numbers
 * 2. We will sort once by every digit, so we will iterate n_iteration times where n_iterations is
 * the number of digits of the largest number in the array
 * 3. Starting with the least significant digit (LSD, the first digit from the right), we iterate
 * the array and push each value to the corresponding dynamic array of the digit. This way we sort
 * by the current digit (counting phase)
 * 4. Iterate all dynamic arrays for digits 0 to 10 in order and write the values back to the
 * original array (collecting phase)
 * 5. Repeat steps 3 and 4 for all N digits to sort the entire array
 */
int *radix_sort(int *array, int N) {
    // Create the bins to put the numbers in to
    vector *vectors[10];
    for (int i = 0; i < 10; ++i)
        vectors[i] = vector_create();

    // Find the max value and make sure all values are positive
    int max_value = -1;
    for (int i = 0; i < N; ++i) {
        assert(array[i] > 0);
        if (array[i] > max_value)
            max_value = array[i];
    }

    // Execute steps 3 and 4
    int n_iterations = digit_count(max_value);
    for (int n = 0; n < n_iterations; ++n) {
        // counting phase
        for (int i = 0; i < N; ++i) {
            int digit = nth_digit(array[i], n);
            vector_push_back(vectors[digit], array[i]);
        }

        // collecting phase
        int current_index = 0;
        for (int i = 0; i < 10; ++i) {
            const int *data = vector_data(vectors[i]);
            const int size = vector_size(vectors[i]);

            for (int j = 0; j < size; ++j)
                array[current_index++] = data[j];

            vector_clear(vectors[i]);
        }
    }

    for (int i = 0; i < 10; ++i)
        vector_free(vectors[i]);

    return array;
}

void test_array_is_sorted() {
    const int N = 6;
    int sorted[] = {1, 2, 5, 7, 11, 1232};
    int not_sorted[] = {1, 5, 2, 7, 11, 1232};

    assert(array_is_sorted(sorted, N));
    assert(!array_is_sorted(not_sorted, N));
}

void test_nth_digit() {
    int number = 1234567890;
    int digits[] = {0, 9, 8, 7, 6, 5, 4, 3, 2, 1};

    for (int i = 0; i < 10; ++i)
        assert(nth_digit(number, i) == digits[i]);
}

void test_vector() {
    vector *vector = vector_create();
    for (int i = 0; i < 10; ++i) {
```

```
84:         vector_push_back(vector, i);
85:         assert(vector_size(vector) == i + 1);
86:     }
87:
88:     for (int i = 0; i < vector_size(vector); ++i)
89:         assert(vector_data(vector)[i] == i);
90:
91:     vector_free(vector);
92: }
93:
94: void test_digit_count() {
95:     int values[] = {1, 0, 12, 123, 10000};
96:     int counts[] = {1, 1, 2, 3, 5};
97:
98:     for (int i = 0; i < 5; ++i)
99:         assert(digit_count(values[i]) == counts[i]);
100: }
101:
102: void test_read_numbers() {
103:     const char *path = "numbers.txt";
104:     vector *vector = read_numbers_from_file(path);
105:
106:     assert(vector_size(vector) > 0);
107:
108:     vector_free(vector);
109: }
110:
111: void test_radix_sort() {
112:     const int N = 100;
113:     int *array = generate_random_array(N);
114:     radix_sort(array, N);
115:     assert(array_is_sorted(array, N));
116:
117:     free(array);
118: }
119:
120: void test() {
121:     printf("Starting tests\n");
122:     test_nth_digit();
123:     test_vector();
124:     test_digit_count();
125:     test_read_numbers();
126:     test_radix_sort();
127:     test_array_is_sorted();
128:
129:     printf("All tests passed\n");
130: }
131:
132: /**
133:  * @brief Reads in numbers.txt and measures the time radixsort takes to sort it
134:  */
135: void benchmark() {
136:     const char *path = "numbers.txt";
137:     vector *vector = read_numbers_from_file(path);
138:     int *array = vector_data(vector);
139:
140:     // I prefer timespec because it supports ns precision
141:     struct timespec start, end;
142:     clock_gettime(CLOCK_MONOTONIC, &start);
143:     radix_sort(array, vector_size(vector));
144:     clock_gettime(CLOCK_MONOTONIC, &end);
145:
146:     assert(array_is_sorted(array, vector_size(vector)));
147:
148:     print_elapsed_time(start, end);
149:
150:     vector_free(vector);
151: }
152:
153: int main() {
154:     srand(time(NULL));
155:
156:     test();
157:     benchmark();
158: }
```

 Moritz Lunk, bt712482 | Yannik Schroeder, bt722248 | Justin Stoll, bt709742

```c
#include "util.h"

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *print_ID(void *) {
    pthread_t threadID = pthread_self();
    printf("Thread ID is %lu\n", (unsigned long)threadID);
    return NULL;
}

int main(int argc, char **argv) {
    if (argc != 2) {
        printf("Usage: %s <number of threads>\n", argv[0]);
        return 0;
    }

    long N;
    convert_to_number(argv[1], &N, LONG);

    pthread_t *threads = (pthread_t *)malloc(N * sizeof(pthread_t));
    abort_on_failed_allocation(threads);

    for (int i = 0; i < N; ++i) {
        int result = pthread_create(&threads[i], NULL, print_ID, NULL);

        if (result != 0) {
            fprintf(stderr, "Error creating thread\n");
            exit(1);
        }
    }

    for (int i = 0; i < N; ++i) {
        int result = pthread_join(threads[i], NULL);

        if (result != 0) {
            fprintf(stderr, "Error joining thread\n");
            exit(1);
        }
    }

    free(threads);

    printf("I started %li threads.\n", N);

    return 0;
}
```

 Moritz Lunk, bt712482 | Yannik Schroeder, bt722248 | Justin Stoll, bt709742

```c
#include "util.h"

#include <assert.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void test_palindrome() {
    // even
    const char *s1 = "otto";
    const char *s2 = "hallo";

    // odd
    const char *s3 = "oto";
    const char *s4 = "asd";

    assert(is_palindrome(s1));
    assert(!is_palindrome(s2));
    assert(is_palindrome(s3));
    assert(!is_palindrome(s4));
}

void *is_palindrome_func(void *arg) {
    const char *str = (const char *)arg;

    pthread_t threadID = pthread_self();
    printf("Thread with ID %lu is now working on string %s\n", (unsigned long)threadID, str);

    if (is_palindrome((const char *)str))
        printf("%s is a palindrome\n", str);
    else
        printf("%s is not a palindrome\n", str);

    return NULL;
}

int main(int argc, char **argv) {
    test_palindrome();

    if (argc < 2) {
        printf("Usage: %s <str1> <str2> ...\n", argv[0]);
        return 1;
    }

    int N = argc - 1;
    pthread_t *threads = (pthread_t *)malloc(N * sizeof(pthread_t));
    abort_on_failed_allocation(threads);

    for (int i = 0; i < N; ++i) {
        int result = pthread_create(&threads[i], NULL, is_palindrome_func, (void *)argv[i + 1]);

        if (result != 0) {
            fprintf(stderr, "Error creating thread\n");
            exit(1);
        }
    }

    for (int i = 0; i < N; ++i) {
        int result = pthread_join(threads[i], NULL);

        if (result != 0) {
            fprintf(stderr, "Error joining thread\n");
            exit(1);
        }
    }

    free(threads);

    printf("Main: Threads joined successfully.\n");

    return 0;
}
```

```c
#ifndef UTIL_H
#define UTIL_H

#include "vector.h"

#include <stdbool.h>
#include <time.h>

/**
 * @brief Generates a random integer between min and max
 */
int rand_range(int min, int max);

/**
 * @brief Generates a random array of length N with random values larger than 0
 */
int *generate_random_array(int N);

/**
 * @returns n-th digit of the given number (from the right)
 */
int nth_digit(int number, int n);

/**
 * @returns Number of digits of the given number
 */
int digit_count(int number);

/**
 * @returns true if the array is sorted in ascending order
 */
bool array_is_sorted(int *array, int N);

/**
 * @brief Reads in space separated numbers from a file
 */
vector *read_numbers_from_file(const char *path);

/**
 * @brief Prints the elapsed time between 2 time points in a reasonable unit
 * @param start First time point
 * @param end Second time point
 */
void print_elapsed_time(struct timespec start, struct timespec end);

/**
 * Types required by the convert_to_number function
 */
typedef enum { LONG, FLOAT } conversion_type;

/**
 * Helper function for error messages
 */
const char *type_to_string(conversion_type type);

/**
 * Checks if the result of a conversion using strtol or strtof was successfull
 * @param str String containing a number (or not)
 * @param end This is the convention:
 * successful conversion - end points to null terminator
 * partial conversion - end points to first invalid character
 * no conversion - end and str are the same
 */
bool conversion_successfull(const char *str, char *end);

/**
 * Converts a given string to a float or long
 * @param str String containing a number (or not)
 * @param result Either a long or float (please)
 * @param type Desired conversion type
 */
void convert_to_number(const char *str, void *result, conversion_type type);

/**
 * @returns true if the given string is a palindrome
 */
bool is_palindrome(const char *str);

/**
 * @brief Calls abort() if the given pointer is NULL and prints a message indicating that memory
 * allocation failed
 */
void abort_on_failed_allocation(void *ptr);
```

```
84:
85: #endif /* UTIL_H */
```

```c
#include "util.h"
#include "vector.h"

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// We're only sorting signed ints, so we don't need values > 2^31
static const int powers_of_10[] = {1,       10,       100,       1000,       10000,
                                   100000, 1000000, 10000000, 100000000, 1000000000};

int rand_range(int min, int max) {
    return rand() % (max - min + 1) + min;
}

int *generate_random_array(int N) {
    int *array = (int *)malloc(N * sizeof(int));

    abort_on_failed_allocation(array);

    for (int i = 0; i < N; ++i)
        array[i] = rand_range(0, 100000);

    return array;
}

int nth_digit(int number, int n) {
    // https://stackoverflow.com/a/203877
    return ((number / powers_of_10[n]) % 10);
}

bool array_is_sorted(int *array, int N) {
    for (int i = 0; i < N - 1; ++i) {
        if (array[i] > array[i + 1])
            return false;
    }

    return true;
}

int digit_count(int number) {
    int count = 1;
    while ((number /= 10) > 0)
        ++count;

    return count;
}

vector *read_numbers_from_file(const char *path) {
    FILE *file = fopen(path, "r");
    if (!file) {
        fprintf(stderr, "Couldn't open file %s", path);
        exit(1);
    }

    vector *vector = vector_create();
    int number;
    while (fscanf(file, "%d", &number) == 1)
        vector_push_back(vector, number);

    fclose(file);

    printf("Read in %d numbers from %s\n", vector_size(vector), path);

    return vector;
}

void print_elapsed_time(struct timespec start, struct timespec end) {
    const double time_ns = (end.tv_sec - start.tv_sec) * 1e9 + (end.tv_nsec - start.tv_nsec);
    const char *time_units[] = {"ns", "us", "ms", "s"};

    int i = 0;
    double converted_time = time_ns;
    while (converted_time > 1e3 && i < (sizeof(time_units) / sizeof(time_units[0])) - 1) {
        converted_time /= 1e3;
        ++i;
    }

    printf("Elapsed time: %lf%s\n", converted_time, time_units[i]);
}

const char *type_to_string(conversion_type type) {
```

```
84:     return type == LONG ? "long" : "float";
85: }
86:
87: bool conversion_successfull(const char *str, char *end) {
88:     return errno != ERANGE && str != end && *end == '\0';
89: }
90:
91: void convert_to_number(const char *str, void *result, conversion_type type) {
92:     if (str == NULL || result == NULL) {
93:         fprintf(stderr, "Can't convert a null pointer to %s\n", type_to_string(type));
94:         exit(1);
95:     }
96:
97:     char *end;
98:     errno = 0; // reset after potential previous call
99:
100:    if (type == LONG) {
101:        *(long *)result = strtol(str, &end, 10);
102:    } else if (type == FLOAT) {
103:        *(float *)result = strtof(str, &end);
104:    } else {
105:        fprintf(stderr, "Unknown type for conversion given\n");
106:        exit(1);
107:    }
108:
109:    if (!conversion_successfull(str, end)) {
110:        fprintf(stderr, "Couldn't convert \"%s\" to %s\n", str, type_to_string(type));
111:        exit(1);
112:    }
113: }
114:
115: bool is_palindrome(const char *str) {
116:     int len = strlen(str);
117:
118:     for (int i = 0; i < len / 2; ++i) {
119:         if (str[i] != str[len - 1 - i])
120:             return false;
121:     }
122:
123:     return true;
124: }
125:
126: void abort_on_failed_allocation(void *ptr) {
127:     if (ptr == NULL) {
128:         fprintf(stderr, "Memory allocation failed\n");
129:         abort();
130:     }
131: }
```

 Moritz Lunk, bt712482 | Yannik Schroeder, bt722248 | Justin Stoll, bt709742

```c
#ifndef VECTOR_H
#define VECTOR_H

/**
 * @brief Simple vector class for dynamic arrays
 */
typedef struct vector vector;

/**
 * @brief Creates a vector of size 0
 */
vector *vector_create();

/**
 * @brief Frees the memory of the vector
 */
void vector_free(vector *vector);

/**
 * @brief Appends a value to the back of the vector
 */
void vector_push_back(vector *vector, int number);

/**
 * @return Number of elements in the vector
 */
int vector_size(const vector *vector);

/**
 * @returns The underlying array used in the vector
 */
int *vector_data(vector *vector);

/**
 * @brief Resets the size of the vector to 0
 */
void vector_clear(vector *vector);

#endif /* VECTOR_H */
```

 Moritz Lunk, bt712482 | Yannik Schroeder, bt722248 | Justin Stoll, bt709742

```c
#include "vector.h"
#include "util.h"

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

struct vector {
    int *data_;
    int capacity_;
    int top_index_;
};

static void vector_grow(vector *vector) {
    int new_capacity = vector->capacity_ * 2;
    int *new_data = realloc(vector->data_, new_capacity * sizeof(int));

    abort_on_failed_allocation(new_data);

    vector->data_ = new_data;
    vector->capacity_ = new_capacity;
}

vector *vector_create() {
    vector *v = (vector *)malloc(sizeof(vector));
    v->capacity_ = 2;
    v->top_index_ = -1;
    v->data_ = (int *)calloc(v->capacity_, sizeof(int));

    abort_on_failed_allocation(v->data_);

    return v;
}

void vector_free(vector *vector) {
    free(vector->data_);
    free(vector);
}

void vector_push_back(vector *vector, int number) {
    if (vector->top_index_ == vector->capacity_ - 1)
        vector_grow(vector);

    ++vector->top_index_;
    vector->data_[vector->top_index_] = number;
}

int *vector_data(vector *vector) {
    return vector->data_;
}

int vector_size(const vector *vector) {
    return vector->top_index_ + 1;
}

void vector_clear(vector *vector) {
    vector->top_index_ = -1;
}
```