SWO3	Übung zu Softwareentwicklung mit klassischen Sprachen und Bibliotheken 3	WS 2018/19, Angabe 4

☐ Gruppe 1 (J. Heinzelreiter)
☐ Gruppe 2 (M. Hava) Name: Niklas Vest Aufwand [h]: 10
☒ Gruppe 3 (P. Kulczycki) Übungsleiter/Tutor: _____ Punkte: _____

Beispiel	Lösungsidee (max. 100%)	Implement. (max. 100%)	Testen (max. 100%)
1 (15P + 35 P)	80%	90%	95%
2 (50 P)	80%	100%	90%

Beispiel 1: Longest Increasing Subsequence (`src/lis`)

Gegeben sei eine beliebig lange, unsortierte Folge von positiven ganzen Zahlen und die Anzahl `n` der Elemente dieser Folge, in C dargestellt durch folgende Definitionen (mit Initialisierungen, die nur als Beispiel dienen):

```
#define MAX 100

                  // 0, 1, 2, 3, 4, 5, 6, 7, 8
int const s [MAX] = {9, 5, 2, 8, 7, 3, 1, 6, 4};
int const n       = 9;   // number of elements in s
```

Der *Longest Increasing Run* (LIR) ist die Länge der längsten zusammenhängenden Teilfolge (engl. *run*) von Elementen, deren Werte streng monoton steigend (engl. *increasing*) sind. Für das obige Beispiel mit den Werten `9,5,2,8,7,3,1,6,4` ergibt sich 2 (die beiden Läufe sind unterstrichen). Dieses Problem kann man in linearer Zeit lösen.

Die Berechnung der *Longest Increasing Subsequence* (LIS) besteht darin, die Länge der längsten nicht zusammenhängenden Teilfolge (engl. *subsequence*) monoton steigender Elemente zu ermitteln. Für das obige Beispiel `9,5,2,8,7,3,1,6,4` ergibt sich hierfür 3, wobei es (rein zufällig) wieder zwei solcher längsten Teilfolgen gibt, nämlich `2,3,6` und `2,3,4`.

Mit dynamischer Programmierung lässt sich auch dieses Problem lösen. Die zentrale Idee: Wenn die LIS für alle Anfänge der Gesamtfolge, also z.B. für `s[1..i-1]` bekannt ist, so ergibt sich die LIS für die um 1 längere Anfangsfolge durch Hinzunahme des Elements `s[i]`, indem die Länge der bisher längsten Teilfolge (also deren Maximum) um 1 erhöht wird, wenn diese mit einem Element `s[j] < s[i]` geendet hat. Diese Erkenntnis kann man in einem Feld `l` für alle Längen der Anfangsfolgen abbilden, dessen Elemente iterativ (für `i = 1..n`) wie folgt berechnet werden:

$$l_i = \max_{1 \le j < i} l_j + 1 \qquad \text{mit } s_j < s_i$$

Um die Elemente der längsten Teilfolge später auch rekonstruieren zu können bietet es sich an, neben dem Feld `l` für auch ein Feld `p` für den Index des jeweiligen Vorgängers (engl. *predecessor*) mitzuführen. Folgende Tabelle zeigt das Ergebnis für obiges Beispiel:

i	0	1	2	3	4	5	6	7	8
s[i]	9	5	2	8	7	3	1	6	4
l[i]	1	1	1	2	2	2	1	3	**3**
p[i]	-	-	-	1	1	2	-	5	5

Entwickeln Sie nun ein C-Programm, das die beiden Funktionen

```
int longest_increasing_run         (int const s [], int const n);
int longest_increasing_subsequence (int const s [], int const n);
```

implementiert. Die `main`-Funktion stellt im Wesentlichen einen Testreiber dar, teilen Sie Ihr C-Programm sinnvoll auf mehrere Module auf.

Beispiel 2: Teambuilding (`src/team/`)

Eine Disziplin in der Leichtathletik ist die 4 x 100-m-Staffel. Für Wettkämpfe (z.B. bei den olympischen Spielen) stellt jedes Teilnehmerland eine Staffel zusammen, die aus den besten LäuferInnen dieses Lands besteht. Gibt es genügend qualifizierte TeilnehmerInnen, so kann es auch noch eine zweite und eine dritte Staffel geben.

Sie sind Bundestrainer einer Leichtathletik-Nation und im Trainingscamp für die 4 x 100-m-Staffel zuständig. Damit Sie Ihre Viererteams auch gegeneinander unter interessanten Bedingungen antreten lassen können, möchten Sie aus den n TeilnehmerInnen (mit bekannten Bestzeiten über 100 m) $n/4$ Viererteams so zusammenstellen, dass die Siegchancen dieser Teams in den Trainingsläufen möglichst gleich sind. Dazu wird das arithmetische Mittel der Bestzeiten der vier Teammitglieder berechnet. Ziel ist es, die Standardabweichung der durchschnittlichen Bestzeiten aller Teams zu minimieren.

Entwickeln Sie einen Exhaustionsalgorithmus mit Optimierung und realisieren Sie diesen in einem C-Programm, das die Gruppeneinteilung errechnet und ausgibt.

Ausarbeitung Übung 04

Table of Contents

Anmerkung....2
Longest Increasing Subsequence....2
Lösungsidee....2
Implementierung....2
main.c....2
Testergebnisse....4
Teambuilding....5
Lösungsidee....5
Implementierung....5
main.c....5
list.h....6
list.c....8
team.h....11
team.c....11
Testergebnisse....15
UNIC....15
unic.h....15

Anmerkung

Um die Verständlichkeit des Codes zu erhöhen, habe ich diesmal meine 80 Spalten pro Zeile mehr ausgenützt als sonst und meinen Variablen und Funktionen längere, hoffentlich einleuchtende Bezeichner gegeben. Darunter leidet die Formattierung im Textdokument und ich bitte den Leser, bei Bedarf den Code im preferierten Editor zu begutachten.

Longest Increasing Subsequence

Lösungsidee

a) **LIR**
Beim Iterieren über die Folge wird ständig mit dem vorhergehenden Wert verglichen.

- Bei aufsteigendem Wert wird aktueller Run-Zähler wird erhöht.
- Bei absteigendem oder gleichem Wert wird der momentane Run beendet.

Ist der neu beobachtete Run länger als der gespeicherte Maximalwert, wird er ersetzt.

b) **LIS**
Für die *Longest Increasing Subsequence* gebe ich keine Lösungsidee an, da der gesamte Lösungsweg ausführlich in der Angabe beschrieben steht. Nur das Hilfsfeld *p* ist nicht weiter erklärt, weswegen ich auch nur diesbezüglich meine Vorgehensweise erläutere. Der Wert von p_i ergibt sich folgendermaßen:

$$p_i = j\Big|_{\substack{max\ p_j \wedge s_j < s_i \\ 1 \le j < i}}$$

Also das größte j, für das gilt, dass p_j der Maximalwert im Intervall *[1, i-1] und* s_j *kleiner als* s_i *ist.*

Implementierung

main.c

```c
#include <stdio.h>
#include "unic.h"

#define MAX 100

int max(int a, int b)
{
    return a > b ? a : b;
}

int longest_increasing_run(int const s[], int const n)
{
    int last_max_run = 0;
    int curr_run = 1;
    int last_num = s[0];

    for (int i = 0; i < n; ++i) {
        if (last_num < s[i]) {
            // run still going
            ++curr_run;
        } else {
            // run ended
            if (last_max_run < curr_run) {
                // run was longer than previous max run
                last_max_run = curr_run;
```

```
            }
            curr_run = 1;
        }
        last_num = s[i];
    }

    // in case the sequence ended with a run
    if (curr_run > last_max_run) {
        last_max_run = curr_run;
    }

    return last_max_run;
}

int longest_increasing_subsequence(int const s[], int const n)
{
    int l[MAX] = {0};
    int p[MAX] = {0};
    for (int i = 0; i < n; ++i) {
        p[i] = -1;
    }
    int max_seq = 0;
    for (int i = 0; i < n; ++i) {
        max_seq = 0;
        for (int j = 1; j < i; ++j) {
            if (s[j] < s[i]) {
                // find the max value of l[0,i - 1]
                max_seq = max(l[j], l[j - 1]);
                if (p[i] == -1) {
                    // if the new p is still unset, set it
                    p[i] = j;
                } else if (p[j] > p[p[i]]) {
                    // else check if the current (j) p is larger
                    // than the p previously stored in p[i]
                    p[i] = j;
                }
            }
        }
        // add + 1 according to formula
        l[i] = max_seq + 1;
    }

    // stores the index of the end of the max sequence
    int seq_end = 0;
    for (int i = 0; i < n; ++i) {
        if (l[i] > max_seq) {
            // also find the longest sequence itself and store its length
            max_seq = l[i];
            seq_end = i;
        }
    }

    // now insert it by traversing p in reverse order
    int *ordered_max_seq = (int *) malloc(sizeof(int) * (size_t) max_seq);
    int reverse_index = max_seq - 1;
    while (seq_end != -1) {
        ordered_max_seq[reverse_index] = s[seq_end];
        --reverse_index;
        seq_end = p[seq_end];
    }

    // print dat sequence
    printf("Max sequence: ");
    for (int i = 0; i < max_seq; ++i) {
        printf("%d,", ordered_max_seq[i]);
    }
    printf("\n");
    free(ordered_max_seq);
```

```
    return max_seq;
}

#define INT_ARR(x) (int[x])

int main()
{
    unic_init();
    unic_ass_eq_i(3, longest_increasing_run(INT_ARR(6) {6, 7, 8, 3, 2, 1}, 6), "Sequence
starting with one LIR");
    unic_ass_eq_i(3, longest_increasing_run(INT_ARR(5) {3, 2, 1, 4, 8}, 5), "Sequence ending
with one LIR");
    unic_ass_eq_i(4, longest_increasing_run(INT_ARR(7) {10, 1, 2, 3, 4, 2}, 6), "Sequence
with one LIR in the middle");
    unic_ass_eq_i(7, longest_increasing_run(INT_ARR(7) {1, 2, 3, 4, 5, 6, 7}, 7), "Sequence
with one LIR = n");
    unic_ass_eq_i(2, longest_increasing_run(INT_ARR(5) {5, 6, 2, 1, 2}, 5), "Sequence with
two equally long runs");
    unic_ass_eq_i(3, longest_increasing_run(INT_ARR(5) {5, 6, 7, 1, 2}, 5), "Sequence with
two runs of diff. sizes");
    unic_ass_eq_i(3, longest_increasing_subsequence(INT_ARR(9) {9, 5, 2, 8, 7, 3, 1, 6, 4},
9), "Exercise");
    unic_ass_eq_i(3, longest_increasing_subsequence(INT_ARR(7) {4, 7, 3, 1, 5, 9, 6 }, 7),
"Sequence with three LISs");
    unic_ass_eq_i(3, longest_increasing_subsequence(INT_ARR(5) {4, 1, 7, 2, 5}, 5), "Sequence
with one LISs");
    unic_ass_eq_i(3, longest_increasing_subsequence(INT_ARR(5) {8, 1, 3, 6, 2}, 5), "Sequence
with one LISs (adjacent elements)");
    return unic_get_results();
}
```

Testergebnisse

```
/home/niklas/Documents/Github/fh-hgb-ws1819/swo/ue04/cmake-build-debug/lis
@ Test "Sequence starting with one LIR" succeeded.
@ Test "Sequence ending with one LIR" succeeded.
@ Test "Sequence with one LIR in the middle" succeeded.
@ Test "Sequence with one LIR = n" succeeded.
@ Test "Sequence with two equally long runs" succeeded.
@ Test "Sequence with two runs of diff. sizes" succeeded.
Max sequence: 2,3,6,
@ Test "Exercise" succeeded.
Max sequence: 3,5,9,
@ Test "Sequence with three LISs" succeeded.
Max sequence: 1,2,5,
@ Test "Sequence with one LISs" succeeded.
Max sequence: 1,3,6,
@ Test "Sequence with one LISs (adjacent elements)" succeeded.
================
UNIC ran 10 tests.
10 tests succeeded.
================

Process finished with exit code 0
```

Teambuilding

Lösungsidee

Das erste Problem, nämlich das auswählen von 4 aus *n* qualifizierten Läufern, wird mittels exhaustive Backtracking gelöst. Jede errechnete Kombination wird in einer einfach verketteten Liste von Teams abgelegt.
Nun gilt es, diejenige Kombination von teams zu suchen, welche die geringste Standardabweichung zum Gesamtdurchschnitt aufweisen. Auch dieses Problem wird mittels Backtracking gelöst. Diesmal aber nicht exhaustiv sondern optimierend. Bei jeder vollen Lösung (i. e. eine Kombination von x Teams, sodass *x* + 1 Teams aufgrund fehlender Läufer nicht mehr möglich ist und kein Läufer zwei mal vorkommt) wird die Standardabweichung der Laufzeiten der Läufer mit “der vorherigen” verglichen. Bei einer niedrigeren Stddev. wird die neue Lösung als besser angesehen. Das backtracking wird so realisiert, dass rekursive Funktion zwei extra Parameter hat, welche der speicherung von neuen Teams und der bisher optimalen Kombination dient.
Damit dieselben Teams nicht zwei mal für eine Kombination in betracht gezogen werden, wird bei jedem neuen rekursiven Aufruf der Zeiger auf die Liste mit allen zur Verfügung stehenden Teams so manipuliert, dass er auf das nächste Team in der Liste zeigt.

Implementierung

main.c

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

#include "list.h"
#include "team.h"

void separator()
{
    printf("====================\n");
}

void run_test_exactly_2_teams()
{
    double metrics[] = { 10.34, 15.6,  12.59, 11.64,
                         14.0,  17.22, 13.1,  20.42  };

    // Implementation note: we actually do not need the exact metrics here, any arbitrary
    // array would do, since the function only uses it for backtracking flags
    list *all_teams = get_all_team_compositions(metrics, 8);

    double average = get_average(metrics, 8);
    list *optimal_teams = get_best_combination_of_teams(all_teams, metrics, 8, average);
    print_list(optimal_teams);
    delete_list(all_teams);
    delete_list(optimal_teams);
}

void run_test_uneven_applications()
{
    double metrics[] = { 10.34, 15.6,  12.59, 11.64,
                         14.0,  17.22, 13.1,  20.42,
                         24.5, 12.52 };
    list *all_teams = get_all_team_compositions(metrics, 10);
    double average = get_average(metrics, 10);
    list *optimal_teams = get_best_combination_of_teams(all_teams, metrics, 10, average);
    print_list(optimal_teams);
    delete_list(all_teams);
    delete_list(optimal_teams);
```

```
}

void run_test_only_1_team()
{
    double metrics[] = { 10.34, 14.0,  12.59, 11.64,
                         15.6 };
    list *all_teams = get_all_team_compositions(metrics, 5);
    double average = get_average(metrics, 5);
    list *optimal_teams = get_best_combination_of_teams(all_teams, metrics, 5, average);
    print_list(optimal_teams);
    delete_list(all_teams);
    delete_list(optimal_teams);
}

void run_test_less_than_4_applications()
{
    double metrics[] = { 10.34, 15.6, 12.5 };
    list *all_teams = get_all_team_compositions(metrics, 3);
    double average = get_average(metrics, 3);
    list *optimal_teams = get_best_combination_of_teams(all_teams, metrics, 3, average);
    print_list(optimal_teams);
    delete_list(all_teams);
    delete_list(optimal_teams);
}

void run_test_invalid_teams_for_best_combination()
{
    double metrics[] = { 10.34, 14.0,  12.59, 11.64,
                         15.6 };
    list *optimal_teams = get_best_combination_of_teams(NULL, metrics, 5, 0);
    print_list(optimal_teams);

    // note that I am still releasing
    // hypothetical memory, since this
    // could very well happen in a real
    // use case.
    delete_list(optimal_teams);
}

int main()
{
    separator();
    run_test_exactly_2_teams();
    separator();
    run_test_uneven_applications();
    separator();
    run_test_only_1_team();
    separator();
    run_test_less_than_4_applications();
    separator();
    run_test_invalid_teams_for_best_combination();
    separator();
    return EXIT_SUCCESS;
}
```

list.h

```
#ifndef UE04_LIST_H
#define UE04_LIST_H

#define TEAM_SIZE 4

/**
 * A node containing a team composition.
 */
typedef struct list_node
{
    /**
     * The next node in the list.
```

```
     */
    struct list_node *next;

    /**
     * The array of team members.
     */
    int *team_composition;
} list_node;

/**
 * A list of team compositions.
 */
typedef struct list
{
    /**
     * The first team in the list.
     */
    list_node *first;

    /**
     * The last team in the list.
     */
    list_node *last;

    /**
     * The size of the container.
     */
    int size;
} list;

/**
 * A list handle is an anchor at
 * which a list is "grabbed". It
 * can grab a list at any element
 * within the list, meaning that all
 * iterative processes using this handle
 * will start at the aforementioned
 * element.
 */
typedef list list_handle;

/**
 * @return A pointer to an empty list.
 * @note Do not forget to call delete_list after usage.
 */
list *create_list();

/**
 * Removes all elements from the list
 * without deleting the entire handle.
 * @param phandle The list to clear.
 */
void clear_list(list_handle *phandle);

/**
 * Deletes the supplied list and all
 * of its content.
 * @param plist The list to delete.
 */
void delete_list(list *plist);

/**
 * Adds a new team composition to the supplied
 * list of teams.
 * @param phandle The list to append to.
 * @param comp The team composition.
 */
void add_team_to_list(list_handle *phandle, const int *comp);
```

```
/**
 * Removes the last team from the list.
 * @param phandle The list to pop().
 */
void pop_list(list_handle *phandle);

 /**
  * Creates a deep copy of the source list and writes it to
  * the destination list.
  * @param destination The list which receives the cloned elements.
  * @param source The source of the clone.
  */
void clone_list(list_handle *destination, const list_handle *source);

/**
 * Creates a new list object with its first element
 * being the supplied list node.
 * @param phandle The list to create a new handle for.
 * @param pnode The node "at which to grab the list".
 * @return A pointer to a list starting with the
 *          supplied <i>node</i>.
 * @note This does not clone the list contents, it just
 *          creates a new anchor. This anchor must be
 *          deleted using {@link delete_list_handle}.
 */
list_handle *get_list_handle(list_handle *phandle, list_node *pnode);

/**
 * Deletes the supplied list handle.
 * @param phandle The handle to delete.
 * @note This does not delete the lists contents.
 */
void delete_list_handle(list_handle *phandle);

/**
 * Prints a list to the standard output.
 * @param phandle The list to print.
 */
void print_list(list_handle *phandle);

#endif //!UE04_LIST_H
```

list.c

```
#include "list.h"
#include <malloc.h>

/**
 * @param comp A team composition.
 * @return A pointer to the newly created
 *          list node containing the supplied
 *          team composition.
 */
static list_node *create_list_node(const int comp[]);

/**
 * Deletes a node.
 * @param node The node to delete.
 */
static void delete_list_node(list_node *pnode);

list *create_list()
{
    list *plist = (list *) malloc(sizeof(list));
    plist->last = NULL;
    plist->first = NULL;
    plist->size = 0;
    return plist;
}
```

```
void delete_list(list *plist)
{
    if (plist != NULL) {
        clear_list(plist);
        delete_list_handle(plist);
    }
}

void add_team_to_list(list_handle *phandle, const int *comp)
{
    if (phandle != NULL) {
        list_node *pnode = create_list_node(comp);
        if (phandle->first == NULL) {
            phandle->first = pnode;
        } else {
            phandle->last->next = pnode;
        }
        phandle->last = pnode;
        phandle->size += 1;
    }
}

void pop_list(list_handle *phandle)
{
    if (phandle != NULL && phandle->size > 0) {
        if (phandle->size == 1) {
            // if there is only one element
            delete_list_node(phandle->first);
            phandle->first = NULL;
            phandle->last = NULL;
        } else { // phandle->size > 1
            // if there are more elements
            list_node *pnode = phandle->first;
            // pnode->next->next is safe here since
            // phandle->size > 1!
            while (pnode->next->next != NULL) {
                pnode = pnode->next;
            }
            delete_list_node(pnode->next);
            pnode->next = NULL;
            // let last point to the previously
            // second-to-last node
            phandle->last = pnode;
        }
        phandle->size -= 1;
#ifdef PARCEL_WRAPPING_POPS
        // I wanted to use a random
        // tab distance but calling
        // rand() a few thousand (pot.
        // million) times wrecked even
        // my HPC-Machine.
        static int tabs = 0;
        for (int i = 0; i < tabs; ++i) {
            printf("\t");
        }
        printf("*pop*\n");
        if (tabs == 5) {
            tabs = 0;
        } else {
            tabs++;
        }
#endif
    }
}

void clone_list(list_handle *destination, const list_handle *source)
{
    if (source != NULL && destination != NULL) {
```

```
        list_node *pnode = source->first;
        while (pnode != NULL) {
            add_team_to_list(destination, pnode->team_composition);
            pnode = pnode->next;
        }
    } else {
        printf("Could not clone list.");
    }
}

list_handle *get_list_handle(list_handle *phandle, list_node *pnode)
{
    list *new_phandle = create_list();
    new_phandle->first = pnode;
    new_phandle->last = phandle→last;

    // calculate new size
    int i = 0;
    list_node *new_pnode = new_phandle->first;
    while (new_pnode != NULL) {
        ++i;
        new_pnode = new_pnode->next;
    }
    new_phandle->size = i;

    return new_phandle;
}

void delete_list_handle(list_handle *phandle)
{
    if (phandle != NULL) {
        free(phandle);
    }
}

void print_list(list_handle *phandle)
{
    if (phandle != NULL) {
        list_node *pnode = phandle->first;
        while (pnode != NULL) {
            int *comp = pnode->team_composition;
            // team size is guaranteed to be 4
            printf("[%d, %d, %d, %d]\n", comp[0], comp[1], comp[2], comp[3]);
            pnode = pnode->next;
        }
    } else {
        printf("(NULL list)\n");
    }
}

void clear_list(list_handle *phandle)
{
    if (phandle != NULL) {
        list_node *pnode = phandle->first;
        while (pnode != NULL) {
            list_node *prev = pnode;
            pnode = pnode->next;
            delete_list_node(prev);
        }
        // reset values
        phandle->first = NULL;
        phandle->last = NULL;
        phandle->size = 0;
    }
}

// ---- PRIVATE FUNCTIONS ---- //
list_node *create_list_node(const int *comp)
{
```

```
    list_node *pnode = (list_node *) malloc(sizeof(list_node));
    pnode->next = NULL;
    pnode->team_composition = (int *) malloc(sizeof(int) * TEAM_SIZE);
    for (int i = 0; i < TEAM_SIZE; ++i) {
        pnode->team_composition[i] = comp[i];
    }
    return pnode;
}

void delete_list_node(list_node *pnode)
{
    free(pnode->team_composition);
    pnode->team_composition = NULL;
    free(pnode);
}
```

team.h

```
#ifndef UE04_TEAM_H
#define UE04_TEAM_H

#include "list.h"

/**
 * @param metrics The array of applications.
 * @param applications The number of applications.
 * @return The list of possible compositions for
 *          teams of 4.
 */
list *get_all_team_compositions(double *metrics, int applications);

/**
 * @param all_teams The list of teams to consider.
 * @param metrics The metrics for all applications.
 * @param applications The number of applications.
 * @param average The average time of all metrics.
 * @return The best possible combination of teams
 *          with a minimum standard deviation to
 *          the average metric.
 */
list *get_best_combination_of_teams(list_handle *all_teams,
                                    const double *metrics,
                                    int applications,
                                    double average);
/**
 * @param metrics The data set.
 * @param applications The number of metrics
 *                      to use for calculation.
 * @return Returns the average of the supplied
 *          list of metrics.
 */
double get_average(const double *metrics, int applications);

#endif //!UE04_TEAM_H
```

team.c

```
#include "team.h"
#include <stdbool.h>
#include <math.h>
#include <float.h>
#include <stddef.h>
#include <stdio.h>

/**
 * @param existing_teams The list of existing teams.
 * @param team The new team.
 * @return True if the new team does not contain
```

```
 *          any members that belong to any of the
 *          existing teams.
 */
static bool is_team_composition_available(list_handle *existing_teams, list_node *team);

/**
 * @param teams The teams who's average metric to compare to
 *              the overall average.
 * @param metrics The metrics of all members of the specified teams.
 * @param average The overall average to use for comparison.
 * @return The standard deviation given the average
 *          metric and a list of teams.
 */
static double get_standard_deviation(list_handle *teams, const double *metrics, double
average);

/**
 * Finds all possible compositions for
 * teams of 4 using the supplied application
 * metrics.
 * @param team_list The list which will contain all teams.
 * @param metrics The array of applications.
 * @param applications The number of applications.
 * @param team_as_arr The currently processed team composition.
 * @param curr_team_pos The currently processed team position.
 */
static void get_all_team_compositions_helper(list_handle *team_list,
                                             double *metrics,
                                             int applications,
                                             int *team_as_arr,
                                             int curr_team_pos);

/**
 *
 * @param all_teams The list of teams to consider.
 * @param new_teams The currently processed new team combination.
 * @param optimal_teams The stored optimum.
 * @param metrics The metrics for all applications.
 * @param applications The number of applications.
 * @param average_time The average time of all metrics.
 */
static void get_best_combination_of_teams_helper(list *all_teams,
                                                 list *new_teams,
                                                 list *optimal_teams,
                                                 const double *metrics,
                                                 int applications,
                                                 double average_time);

double get_average(const double *metrics, int applications)
{
    double sum = 0;
    for (int i = 0; i < applications; ++i) {
        sum += metrics[i];
    }
    return sum / applications;
}

list *get_all_team_compositions(double *metrics, int applications)
{
    list *plist = NULL;
    if (applications > 3) {
        plist = create_list();
        get_all_team_compositions_helper(plist, metrics, applications, (int[TEAM_SIZE]) {0},
0);
    } else {
        printf("Cannot create teams: A minimum of 4 applications is required to form a team!\
n");
    }
    return plist;
```

```
}

list *get_best_combination_of_teams(list_handle *all_teams, const double *metrics, int
applications, double average)
{
    list *optimal_teams = NULL;
    if (applications > 3 && all_teams != NULL) {
        list *new_teams = create_list();
        optimal_teams = create_list();
        get_best_combination_of_teams_helper(all_teams, new_teams, optimal_teams, metrics,
applications, average);
        delete_list(new_teams);
    } else if (applications <= 3) {
        printf("Cannot find best team combination: A minimum of 4 applications is required to
form a team!\n");
    } else {
        // maybe supplied a valid number of applications but an invalid list of teams
        printf("Cannot find best team combination: Source set is empty.\n");
    }
    return optimal_teams;
}

// ---- PRIVATE FUNCTIONS ---- //
void get_all_team_compositions_helper(list_handle *team_list,
                                      double *metrics,
                                      int applications,
                                      int *team_as_arr,
                                      int curr_team_pos)
{
    for (int current_application = 0; current_application < applications; +
+current_application) {
        if (metrics[current_application] > 0) {

            // a value is accepted for a partial solution
            // by removing its metric from the array (i. e.
            // setting it to 0) and adding the index of the
            // metric to the team array.
            double current_application_metric = metrics[current_application];
            metrics[current_application] -= current_application_metric;
            team_as_arr[curr_team_pos] = current_application;

            // teams of 4 -> 3 is last index!
            if (curr_team_pos == 3) {
                add_team_to_list(team_list, team_as_arr);
            } else {
                // we only need to change curr_team_pos since accepted applications (fixed
values)
                // are denoted by their metric being 0!
                get_all_team_compositions_helper(team_list, metrics, applications,
team_as_arr, curr_team_pos + 1);
            }

            // the backtracking part: add the metric back to the application for future uses
            metrics[current_application] += current_application_metric;
        }
    }
}

void get_best_combination_of_teams_helper(list *all_teams,
                                          list *new_teams,
                                          list *optimal_teams,
                                          const double *metrics,
                                          int applications,
                                          double average_time)
{
    list_node *pteam = all_teams->first;
    // for all remaining teams
    while (pteam != NULL) {
        if (is_team_composition_available(new_teams, pteam)) {
```

```
            add_team_to_list(new_teams, pteam→team_composition);

            if (new_teams->size * 4 + 4 > applications) { // if we can not form another team
of 4
                double stddev_new = get_standard_deviation(new_teams, metrics, average_time);
                double stddev_optimal = get_standard_deviation(optimal_teams, metrics,
average_time);
                if (stddev_new < stddev_optimal) { // if we found a new optimum
                    clear_list(optimal_teams);
                    clone_list(optimal_teams, new_teams);
                }
            } else {
                // if we still have enough applications to form
                // another team (>= 4 applic.), create a new handle
                // starting from team i + 1. That way the next recursion
                // level will not see the current meaning it won't be
                // considered as additional team composition.
                list *new_all_teams = get_list_handle(all_teams, pteam->next);
                get_best_combination_of_teams_helper(
                        new_all_teams, new_teams, optimal_teams, metrics, applications,
average_time);
                delete_list_handle(new_all_teams);
            }

            // backtracking by removing the most recently added
            // team from the list.
            pop_list(new_teams);
        }
        pteam = pteam->next;
    }
}

double get_standard_deviation(list_handle *teams, const double *metrics, double average)
{
    if (teams == NULL || teams->size == 0) {
        // since this functions is only used for finding an optimum
        // invalid calls return the max double, meaning that the
        // check for a new optimum will succeed. For more details,
        // see get_best_combination_of_teams_helper
        return DBL_MAX;
    }
    double average_deviation_sum = 0;
    list_node *pnode = teams->first;
    while (pnode != NULL) {
        double sum = 0;
        for (int i = 0; i < 4; ++i) {
            // team_composition holds indices to the metrics array
            sum += metrics[pnode->team_composition[i]];
        }
        average_deviation_sum += fabs(average - (sum / 4));
        pnode = pnode->next;
    }
    return average_deviation_sum / teams->size;
}

bool is_team_composition_available(list_handle *existing_teams, list_node *team)
{
    bool available = true;
    // for each new team member
    for (int i = 0; i < 4; ++i) {
        int curr_dude = team->team_composition[i];
        // check if he is already in any of the other teams
        list_node *n = existing_teams->first;
        while (n != NULL && available) {
            for (int member = 0; member < 4; ++member) {
                available = available && curr_dude != n->team_composition[member];
            }
            n = n->next;
        }
```

```
    }
    return available;
}
```

Testergebnisse

```
/home/niklas/Documents/Github/fh-hgb-ws1819/swo/ue04/cmake-build-debug/team
==================
[0, 4, 7, 2]
[1, 3, 5, 6]
==================
[0, 4, 8, 9]
[1, 3, 6, 7]
==================
[0, 1, 4, 3]
==================
Cannot create teams: A minimum of 4 applications is required to form a team!
Cannot find best team combination: A minimum of 4 applications is required to form a team!
(NULL list)
==================
Cannot find best team combination: Source set is empty.
(NULL list)
==================

Process finished with exit code 0
```

UNIC

Ich verwende für die Tests in Aufgabe 1 eine eigene kleine Unit-Test Bibliothek, welche ich auch auf Github zur Verfügung stelle.

unic.h

```c
/**
 * The header is from my public gist:
 * https://gist.github.com/YourPsychiatrist/ab56d2879d51289a4170196a3b57b5d9
 * I am the rightful owner of this code
 */
#ifndef UNIC_H
#define UNIC_H
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/*
 * PUBLIC INTERFACE
 */

/**
 * Initializes the unic library
 */
void unic_init();

/**
 * Prints test results. (Only relevant if
 * unic_init was called before running tests!)
 * @return EXIT_SUCCESS if all tests succeeded,
 *          EXIT_FAILURE otherwise.
 */
int unic_get_results();

/**
 * Succeeds if the supplied bool evaluates to true.
 * @param got The value to check.
 * @param name The name of the test.
 */
```

```c
void unic_ass_true(bool got, const char *name);

/**
 * Succeeds if the supplied bool evaluates to false.
 * @param got The value to check.
 * @param name The name of the test.
 */
void unic_ass_false(bool got, const char *name);

/**
 * Succeeds if the supplied values are equal.
 * @param a One of the values to compare.
 * @param b The other value.
 * @param name The name of the test.
 */
void unic_ass_eq_b(bool a, bool b, const char *name);

/**
 * Succeeds if the supplied values are not equal.
 * @param a One of the values to compare.
 * @param b The other value.
 * @param name The name of the test.
 */
void unic_ass_not_eq_b(bool a, bool b, const char *name);

/**
 * Succeeds if the supplied values are equal.
 * @param a One of the values to compare.
 * @param b The other value.
 * @param name The name of the test.
 */
void unic_ass_eq_i(int a, int b, const char *name);

/**
 * Succeeds if the supplied values are not equal.
 * @param a One of the values to compare.
 * @param b The other value.
 * @param name The name of the test.
 */
void unic_ass_not_eq_i(int a, int b, const char *name);

/**
 * Succeeds if the supplied values are equal.
 * @param a One of the values to compare.
 * @param b The other value.
 * @param name The name of the test.
 */
void unic_ass_eq_str(const char *a, const char *b, const char *name);

/**
 * Succeeds if the supplied values are not equal.
 * @param a One of the values to compare.
 * @param b The other value.
 * @param name The name of the test.
 */
void unic_ass_not_eq_str(const char *a, const char *b, const char *name);

/**
 * Succeeds if the supplied values are equal.
 * @param a One of the values to compare.
 * @param b The other value.
 * @param name The name of the test.
 */
void unic_ass_eq_f(double a, double b, const char *name);

/**
 * Succeeds if the supplied values are not equal.
 * @param a One of the values to compare.
 * @param b The other value.
```

```
 * @param name The name of the test.
 */
void unic_ass_not_eq_f(double a, double b, const char *name);

/*
 * "PRIVATE INTERFACE"
 */

// A list of colors for the terminal.

const char * UNIC_COLOR_RED = "\x1b[31m";
const char * UNIC_COLOR_GREEN = "\x1b[32m";
const char * UNIC_COLOR_CYAN = "\x1b[36m";
const char * UNIC_COLOR_RESET = "\x1b[0m";

// Formatting strings

const char *UNIC_EXPECTED_INDENT = "\t|-";
const char *UNIC_GOT_INDENT = "\t'-";

/**
 * The compound holding test results.
 */
struct
{
    unsigned long long succeeded_tests;
    unsigned long long failed_tests;
} global_results;

/**
 * A generic message printing whether a test
 * succeeded.
 * @param success The success flag.
 * @param name The name of the test.
 */
void unic_print_success(bool success, const char * name);

/**
 * Prints the difference between the two
 * arguments a and b.
 */
void unic_print_diff_b(bool a, bool b);

/**
 * Prints the difference between the two
 * arguments a and b.
 */
void unic_print_diff_i(int a, int b);

/**
 * Prints the difference between the two
 * arguments a and b.
 */
void unic_print_diff_str(const char *a, const char *b);

/**
 * Prints the difference between the two
 * arguments a and b.
 */
void unic_print_diff_f(double a, double b);

/**
 * Prints a mismatch message for index i.
 * @param i The index at which a mismatch
 *          happened.
 */
void unic_print_diff_array(size_t i);

/**
```

```
 * Prints a message saying that the
 * arrays differ in size.
 */
void unic_print_array_diff_size();

/**
 * Prints a message saying that the
 * arrays are equal.
 */
void unic_print_array_eq();

/*
 * IMPLEMENTATION
 */

void unic_init()
{
    global_results.succeeded_tests = 0;
    global_results.failed_tests = 0;
}

int unic_get_results()
{
    printf("==================\n");
    printf("UNIC ran %llu tests.\n", global_results.succeeded_tests +
global_results.failed_tests);
    if (global_results.succeeded_tests != 0) {
        printf("%llu tests %ssucceeded%s.\n", global_results.succeeded_tests,
UNIC_COLOR_GREEN, UNIC_COLOR_RESET);
    }
    if (global_results.failed_tests != 0) {
        printf("%llu tests %sfailed%s.\n", global_results.failed_tests, UNIC_COLOR_RED,
UNIC_COLOR_RESET);
    }
    printf("==================\n");
    return global_results.failed_tests == 0 ? EXIT_SUCCESS : EXIT_FAILURE;
}

void unic_print_success(bool success, const char * name)
{
    if (success) {
        ++global_results.succeeded_tests;
    } else {
        ++global_results.failed_tests;
    }
    const char *color = success ? UNIC_COLOR_GREEN : UNIC_COLOR_RED;
    printf("@ Test \"%s%s%s\" ", UNIC_COLOR_CYAN, name, UNIC_COLOR_RESET); // print name
    printf("%s%s%s.\n", color, success ? "succeeded" : "FAILED", UNIC_COLOR_RESET); // print
status
}

void unic_ass_true(bool got, const char *name)
{
    bool success = got;
    unic_print_success(success, name);
    if (!success) {
        printf("%s Assertion failed, argument was \"false\".\n", UNIC_GOT_INDENT);
    }
}

void unic_ass_false(bool got, const char *name)
{
    bool success = !got;
    unic_print_success(success, name);
    if (!success) {
        printf("%s Assertion failed, argument was \"true\".\n", UNIC_GOT_INDENT);
    }
}
```

```c
void unic_print_diff_b(bool a, bool b)
{
    printf("%s Arg 1: %d\n%s Arg 2: %d\n", UNIC_EXPECTED_INDENT, a, UNIC_GOT_INDENT, b);
}

void unic_ass_eq_b(bool a, bool b, const char *name)
{
    bool success = a == b;
    unic_print_success(success, name);
    if (!success) {
        unic_print_diff_b(a, b);
    }
}

void unic_ass_not_eq_b(bool a, bool b, const char *name)
{
    bool success = a != b;
    unic_print_success(success, name);
    if (!success) {
        unic_print_diff_b(a, b);
    }
}

void unic_print_diff_i(int a, int b)
{
    printf("%s Arg 1: %d\n%s Arg 2: %d\n", UNIC_EXPECTED_INDENT, a, UNIC_GOT_INDENT, b);
}

void unic_ass_eq_i(int a, int b, const char *name)
{
    bool success = a == b;
    unic_print_success(success, name);
    if (!success) {
        unic_print_diff_i(a, b);
    }
}

void unic_ass_not_eq_i(int a, int b, const char *name)
{
    bool success = a != b;
    unic_print_success(success, name);
    if (!success) {
        unic_print_diff_i(a, b);
    }
}

void unic_print_diff_str(const char *a, const char *b)
{
    printf("%s Arg 1: %s\n%s Arg 2: %s\n", UNIC_EXPECTED_INDENT, a, UNIC_GOT_INDENT, b);
}

void unic_ass_eq_str(const char *a, const char *b, const char *name)
{
    bool success = strcmp(a, b) == 0;
    unic_print_success(success, name);
    if (!success) {
        unic_print_diff_str(a, b);
    }
}

void unic_ass_not_eq_str(const char *a, const char *b, const char *name)
{
    bool success = strcmp(a, b) != 0;
    unic_print_success(success, name);
    if (!success) {
        unic_print_diff_str(a, b);
    }
}
```

```c
void unic_print_diff_f(double a, double b)
{
    printf("%s Arg 1: %f\n%s Arg 2: %f\n", UNIC_EXPECTED_INDENT, a, UNIC_GOT_INDENT, b);
}

void unic_ass_eq_f(double a, double b, const char *name)
{
    bool success = a == b;
    unic_print_success(success, name);
    if (!success) {
        unic_print_diff_f(a, b);
    }
}

void unic_ass_not_eq_f(double a, double b, const char *name)
{
    bool success = a != b;
    unic_print_success(success, name);
    if (!success) {
        unic_print_diff_f(a, b);
    }
}

/*
 * Array assertions are implemented as macros
 * because C doesn't support generic functions
 * and I am too lazy to write array functions
 * for all data types covered by unic.
 */

void unic_print_diff_array(size_t i)
{
    printf("%s Arrays differ at index %lu\n", UNIC_GOT_INDENT, i);
}

void unic_print_array_diff_size()
{
    printf("%s The arrays have different lengths.\n", UNIC_GOT_INDENT);
}

void unic_print_array_eq()
{
    printf("%s Arrays are equal.\n", UNIC_GOT_INDENT);
}

#define unic_array_size(x) \
    (sizeof(x) / sizeof((x)[0]))

#define unic_ass_array_eq(a1, a2, name)                              \
    if (unic_array_size(a1) != unic_array_size(a2)) {                \
        unic_print_success(false, name);                             \
        unic_print_array_diff_size();                                \
    } else {                                                         \
        size_t i = 0;                                                \
        while (i < unic_array_size(a1) && (a1)[i] == (a2)[i]) ++i;   \
        unic_print_success(i == unic_array_size(a1), name);          \
        if (i != unic_array_size(a1)) { unic_print_diff_array(i); } \
    }

#define unic_ass_array_not_eq(a1, a2, name)                          \
    if (unic_array_size(a1) != unic_array_size(a2)) {                \
        unic_print_success(true, name);                              \
    } else {                                                         \
        size_t i = 0;                                                \
        while (i < unic_array_size(a1) && (a1)[i] == (a2)[i]) ++i;   \
        unic_print_success(i != unic_array_size(a1), name);          \
        if (i == unic_array_size(a1)) { unic_print_array_eq(); }     \
    }
#endif // !UNIC_H
```