

☒ Gruppe 1 (J. Heinzlreiter)☐ Gruppe 2 (M. Hava)Name: PAPESH KonstantinAufwand [h]: 10☐ Gruppe 3 (P. Kulczycki)

Übungsleiter/Tutor: _____

Punkte: _____

Beispiel	Lösungsidee (max. 100%)	Implement. (max. 100%)	Testen (max. 100%)
1 (15P + 35 P)	100	100	100
2 (50 P)	50	20	0

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:

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

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.

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

Übungsabgabe 4

Konstantin Papesh

4. November 2018

4.1 Longest Increasing Subsequence

4.1.1 Lösungsidee

Es sind zwei Funktionen zu erstellen, zum einen *Longest Increasing Run* (LIR) und *Longest Increasing Subsequence* (LIS).

Longest Increasing Run

Es soll die längste, aufeinanderfolgende, streng monoton steigende Folge ausgegeben werden. Dieses Problem kann über eine einfache Iteration über das gesamte Feld gelöst werden. Dabei wird eine Schleife von $i = 1..n$ gestartet und das momentane Element mit dem vorherigen verglichen. Ist das momentane Element größer, wird die aktuelle Folge um 1 erhöht. Ist es gleich oder kleiner ist die Folge zu Ende. Dann wird verglichen, ob die abgelaufene Folge länger ist als die momentane längste. Ist dies der Fall, wird der Wert der aktuellen Folge gespeichert. Daraufaufgehend wird die Folge wieder auf 1¹ zurückgesetzt und ein Element weitergegangen.

Longest Increasing Subsequence

Hier soll die längste, nicht bedingt aufeinanderfolgende, streng monoton steigende Folge ausgegeben werden. Diese ist über das ganze Feld zu erstellen. Deren Ermittlung ist in tiefgehendem Detail in der Angabe beschrieben, eine erneute Erläuterung wäre redundant. Lediglich der Algorithmus für $p[i]$ ist nicht definiert, dieses Feld enthält den Index der nächst niedrigeren Zahl der Folge. Dabei wird beim aktuellen Index über alle Elemente zwischen $1 \leq j < i$ gegangen und verglichen, ob der ausgewählte Wert an j kleiner ist als der momentane Wert an i . Wenn dies zutrifft, wird kontrolliert ob $l[j]$ größer ist als das momentan größte l . Also wir suchen uns die bislang längste Folge heraus. Wenn dies zutrifft wird der $tempL$ auf diese Länge gesetzt und der Index, wo dieses $l[j]$ auftritt gespeichert. Ist die Schleife abgelaufen wird das gespeicherte $l[j]$ um 1 erhöht, da die Folge ja fortgesetzt wird. Und p an der Stelle i wird auf j gesetzt, dem nächsten

¹Weil jede Zahl mit sich selbst eine Folge mit 1 ergibt.

Index der Folge. Ist das momentane $l[j]$ das größte l , wird der Index j in einer weiteren Variable gespeichert. Dies ermöglicht es am Ende einfach die längste Folge auszugeben. Dies geschieht, indem man sich von $p[0]$ zu $p[n]$ iteriert.

4.1.2 Implementierung

Listing 4.1: main.c

```

1  #include <stdio.h>
2
3  #define MAX 100
4
5  int longest_increasing_run(int const s[], int const n){
6      int longestRun = 0;
7      int currentRun = 1;
8      for (int i = 1; i < n; ++i) {
9          if(s[i] > s[i-1]){
10             currentRun++;
11         } else {
12             if(currentRun > longestRun)
13                 longestRun = currentRun;
14             currentRun = 1;
15         }
16     }
17     if(currentRun > longestRun)
18         longestRun = currentRun;
19     return longestRun;
20 }
21
22 void printArray(int const s[], int const n, char const name[]) {
23     for (int i = 0; i < n; ++i) {
24         printf("%s[%i] = %i \n", name, i, s[i]);
25     }
26     printf("\n\n");
27 }
28
29 void printSequence(int const s[], int const l[], int const p[], int const maxLIndex)
30 {
31     int tSequence[MAX];
32     int nextIndex = maxLIndex;
33     for (int m = 0; m < l[maxLIndex]; ++m) {
34         tSequence[m] = s[nextIndex];
35         nextIndex = p[nextIndex];
36     }
37     for (int m = l[maxLIndex]-1; m >= 0; --m) {
38         if(m == l[maxLIndex]-1)
39             printf("%i ", tSequence[m]);
40         else
41             printf("- %i ", tSequence[m]);
42     }
43 }
44
45 int longest_increasing_subsequence(int const s[], int const n){
46     int l[MAX];
47     int p[MAX];
48     int maxLIndex = 0;
49     for (int i = 0; i < n; ++i) {
50         int tempL = 1;
51         int tempP = -1;
52         for (int j = 0; j < i; ++j) {

```

```

53         if(s[i] > s[j]){
54             if(l[j] >= tempL){
55                 tempL = l[j]+1;
56                 tempP = j;
57             }
58         }
59     }
60     l[i] = tempL;
61     p[i] = tempP;
62     if(l[i] >= l[maxLIndex]) {
63         maxLIndex = i;
64     }
65 }
66 printArray(s, n, "s");
67 printArray(l, n, "l");
68 printArray(p, n, "p");
69 printSequence(s, l, p, maxLIndex);
70 printf("\n");
71 return l[maxLIndex];
72 }
73
74 int main() {
75     //int const s[MAX] = {1,2,3,4,5,6,7,8,9};
76     //int const s[MAX] = {9,8,7,6,5,4,3,2,1};
77     int const s[MAX] = {0,0,0,0,0,0,0,0,0};
78     int const n = 9;
79
80     printf("Longest increasing run = %i \n\n", longest_increasing_run(s, n));
81     printf("Longest increasing sequence = %i \n\n", longest_increasing_subsequence
82           (s, n));
83     return 0;
84 }

```

4.1.3 Testen

```

Longest increasing run = 2

s[0] = 9
s[1] = 5
s[2] = 2
s[3] = 8
s[4] = 7
s[5] = 3
s[6] = 1
s[7] = 6
s[8] = 4

l[0] = 1
l[1] = 1
l[2] = 1
l[3] = 2
l[4] = 2
l[5] = 2
l[6] = 1
l[7] = 3
l[8] = 3

p[0] = 0
p[1] = 0
p[2] = 0
p[3] = 1
p[4] = 1
p[5] = 2
p[6] = 0
p[7] = 5
p[8] = 5

2 - 3 - 4
Longest increasing sequence = 3

```

Abbildung 4.1: Ausgabe bei einem Feld wie in der Angabe

```

Longest increasing run = 9

s[0] = 1
s[1] = 2
s[2] = 3
s[3] = 4
s[4] = 5
s[5] = 6
s[6] = 7
s[7] = 8
s[8] = 9

l[0] = 2
l[1] = 2
l[2] = 3
l[3] = 4
l[4] = 5
l[5] = 6
l[6] = 7
l[7] = 8
l[8] = 9

p[0] = 0
p[1] = 0
p[2] = 1
p[3] = 2
p[4] = 3
p[5] = 4
p[6] = 5
p[7] = 6
p[8] = 7

1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9
Longest increasing sequence = 9

```

Abbildung 4.2: Ausgabe bei einem stetig ansteigendem Feld

```

Longest increasing run = 1

s[0] = 9
s[1] = 8
s[2] = 7
s[3] = 6
s[4] = 5
s[5] = 4
s[6] = 3
s[7] = 2
s[8] = 1

l[0] = 1
l[1] = 1
l[2] = 1
l[3] = 1
l[4] = 1
l[5] = 1
l[6] = 1
l[7] = 1
l[8] = 1

p[0] = -1
p[1] = -1
p[2] = -1
p[3] = -1
p[4] = -1
p[5] = -1
p[6] = -1
p[7] = -1
p[8] = -1

1
Longest increasing sequence = 1

```

Abbildung 4.3: Ausgabe bei einem stetig fallendem Feld

```

Longest increasing run = 1

s[0] = 0
s[1] = 0
s[2] = 0
s[3] = 0
s[4] = 0
s[5] = 0
s[6] = 0
s[7] = 0
s[8] = 0

l[0] = 1
l[1] = 1
l[2] = 1
l[3] = 1
l[4] = 1
l[5] = 1
l[6] = 1
l[7] = 1
l[8] = 1

p[0] = -1
p[1] = -1
p[2] = -1
p[3] = -1
p[4] = -1
p[5] = -1
p[6] = -1
p[7] = -1
p[8] = -1

0
Longest increasing sequence = 1

```

Abbildung 4.4: Ausgabe bei einem Feld mit identen Werten

4.2 Teambuilding

4.2.1 Lösungsidee

Ziel ist es, ein Programm zu erstellen, welches n 4er-Team zusammenstellt, deren durchschnittliche Zeiten die geringste Standardabweichung haben. Dabei wird zuerst mithilfe einer Funktion alle möglichen Teamkonstellationen erstellt und deren Durchschnittszeit berechnet. Danach werden alle Teams miteinander verglichen, welche eine möglichst idente Durchschnittszeit haben und keine doppelten Personen enthalten. Diese Teamkonstellation wird dann ausgegeben.

4.2.2 Implementierung

Listing 4.2: main.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <stdbool.h>
5 #include <limits.h>
6
7 #define TEAM_SIZE 4
8 #define ARGUMENTS 1
9 #define MAX 1000
10
11 typedef struct runner runner;
12 typedef struct team team;
13
14 void sortRunnerArray(runner *runnerArray[], int length);
15
16 struct runner {
17     int ID;
18     float bestTime;
19 };
20
21 struct team {
22     int ID;
23     runner *runners[TEAM_SIZE];
24     double time;
25 };
26
27 bool checkArgs(int argc, char* argv[]){
28     int amountRunners = argc-ARGUMENTS;
29     if(argc == 1){
30         printf("Usage: %s [runner_time]\n", argv[0]);
31         printf("Example: %s 10.5 10.3 ... \n", argv[0]);
32         return false;
33     }
34     if((amountRunners%TEAM_SIZE) != 0){
35         printf("Amount of runners (%i) must be divideable by the team size = %i",
36             amountRunners, TEAM_SIZE);
37         return false;
38     }
39     return true;
40 }

```

```

40
41 void initRunnerArray(runner *runnerArray[], int length){
42     for (int i = 0; i < length; ++i) {
43         runner* runner = malloc(sizeof(*runner));
44         runnerArray[i] = runner;
45     }
46 }
47
48 void copyRunnerArray(runner *from[], runner *to[], int length){
49     for (int i = 0; i < length; ++i) {
50         to[i] = from[i];
51     }
52 }
53
54 void copyTeamArray(team from[], team to[], int length){
55     for (int i = 0; i < length; ++i) {
56         to[i] = from[i];
57     }
58 }
59
60 void calcSum(team teamArray[], int index) {
61     teamArray[index].time = 0;
62     for (int i = 0; i < TEAM_SIZE; ++i) {
63         teamArray[index].time += teamArray[index].runners[i]->bestTime;
64     }
65 }
66
67 void printTeamArray(team teamArray[], int length){
68     for (int i = 0; i < length; ++i) {
69         for (int j = 0; j < TEAM_SIZE; ++j) {
70             printf("Team %i - Runner %i\n", teamArray[i].ID, teamArray[i].runners[j]
71                 ]->ID);
72             printf("With time: %f\n\n", teamArray[i].time);
73         }
74 }
75
76 int removeRunnerFromArray(runner *from[], runner *to[], int index, int length){
77     int toLength = length-1;
78     copyRunnerArray(from, to, length);
79     to[index] = from[length-1];
80     sortRunnerArray(to, toLength);
81     return toLength;
82 }
83
84 void fillRunnerArrayArg(runner *runnerArray[], char* argv[], int argc){
85     initRunnerArray(runnerArray, argc-ARGUMENTS);
86     for(int i = 0; i < argc-ARGUMENTS; i++){
87         runnerArray[i]->ID = i;
88         runnerArray[i]->bestTime = atof(argv[i+ARGUMENTS]);
89     }
90 }
91
92 void initTeamArray(team *teamArray, int amountTeams){
93     for(int i = 0; i < amountTeams; i++) {
94         teamArray[i].ID = i;
95         for (int j = 0; j < TEAM_SIZE; ++j) {

```

```

96         teamArray[i].runners[j] = NULL;
97     }
98     teamArray[i].time = 0;
99 }
100 }
101
102 void sortRunnerArray(runner *runnerArray[], int length){
103     if(length <= 1)
104         return;
105     runner *left[MAX];
106     runner *right[MAX];
107     int leftMax = ceil((double)length/2);
108     int rightMax = floor((double)length/2);
109
110     for(int i = 0; i < leftMax; i++)
111         left[i] = runnerArray[i];
112     for(int i = 0; i < rightMax; i++)
113         right[i] = runnerArray[leftMax + i];
114
115     sortRunnerArray(left, leftMax);
116     sortRunnerArray(right, rightMax);
117
118     int leftIndex = 0;
119     int rightIndex = 0;
120
121     for(int i = 0; i < length; i++) {
122         if (leftIndex >= leftMax) { //left array fully in a[]
123             runnerArray[i] = right[rightIndex];
124             rightIndex++;
125         } else if (rightIndex >= rightMax) { //right array fully in a[]
126             runnerArray[i] = left[leftIndex];
127             leftIndex++;
128         } else {
129             if (left[leftIndex]->bestTime > right[rightIndex]->bestTime) {
130                 runnerArray[i] = left[leftIndex];
131                 leftIndex++;
132             } else {
133                 runnerArray[i] = right[rightIndex];
134                 rightIndex++;
135             }
136         }
137     }
138 }
139
140 void getAllPossibleTeams(runner *runnerArray[], team possibleTeams[], int
    AMOUNT_RUNNERS, int* amount_teams) {
141     runner *tRunnerArray[MAX];
142     team tCurTeam;
143
144     for (int j = 0; j < TEAM_SIZE; ++j) {
145         tCurTeam.runners[j] = NULL;
146     }
147     tCurTeam.time = 0;
148
149     for (int i = 0; i < AMOUNT_RUNNERS; ++i) {
150         bool success = false;
151         runner* currentRunner = runnerArray[i];

```

```

152     int tAmountRunners = removeRunnerFromArray(runnerArray, tRunnerArray, i,
    AMOUNT_RUNNERS);
153     for (int j = 0; j < TEAM_SIZE; ++j) {
154         if(tCurTeam.runners[j] == NULL) {
155             tCurTeam.runners[j] = currentRunner;
156             success = true;
157         }
158     }
159     if(success)
160         getAllPossibleTeams(tRunnerArray, possibleTeams, tAmountRunners,
    amount_teams);
161     else{//TEAM is full
162         possibleTeams[*amount_teams+1] = tCurTeam;
163         *amount_teams++;
164         return;
165     }
166 }
167 }
168
169 void getBestDistribution(runner *runnerArray[], team teamArray[], int AMOUNT_RUNNERS
    , int AMOUNT_TEAMS){
170     team tTeamArray[MAX];
171     double bestDiff = LONG_MAX;
172     team bestTeamDist[MAX];
173
174     for (int i = 0; i < 1000; ++i) {
175         double sumDiff = 0;
176         copyTeamArray(teamArray, tTeamArray, AMOUNT_TEAMS);
177         //SWAP TEAMS
178         for (int j = 0; j < AMOUNT_TEAMS; ++j) {
179             sumDiff += tTeamArray[j].time;
180         }
181         if(sumDiff < bestDiff) {
182             bestDiff = sumDiff;
183             copyTeamArray(tTeamArray, bestTeamDist, AMOUNT_TEAMS);
184         }
185     }
186 }
187
188 int main(int argc, char* argv[]) {
189     if(!checkArgs(argc, argv)){
190         return EXIT_FAILURE;
191     }
192
193     const int AMOUNT_RUNNERS = argc-ARGUMENTS;
194     const int AMOUNT_TEAMS = AMOUNT_RUNNERS / TEAM_SIZE;
195     runner *runnerArray[MAX];
196     team teamArray[MAX];
197
198     fillRunnerArrayArg(runnerArray, argv, argc);
199     initTeamArray(teamArray, AMOUNT_TEAMS);
200
201     sortRunnerArray(runnerArray, AMOUNT_RUNNERS);
202
203     //getBestDistribution(runnerArray, teamArray, AMOUNT_RUNNERS,
    AMOUNT_TEAMS);
204     team possibleTeams[MAX];

```

```

205     int amountTeams = 0;
206     getAllPossibleTeams(runnerArray, possibleTeams, AMOUNT_RUNNERS, &amountTeams);
207     return EXIT_SUCCESS;
208 }

```

4.2.3 Testen

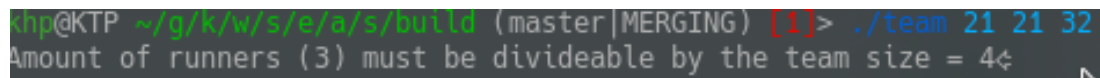


```

khp@KTP ~/g/k/w/s/e/a/s/build (master|MERGING)> ./team
Usage: ./team [runner_time]
Example: ./team 10.5 10.3 ...

```

Abbildung 4.5: Fehlermeldung, wenn keine Argumente mitgegeben



```

khp@KTP ~/g/k/w/s/e/a/s/build (master|MERGING) [1]> ./team 21 21 32
Amount of runners (3) must be divideable by the team size = 4

```

Abbildung 4.6: Fehlermeldung, wenn eine nicht durch 4 dividierbare Anzahl an Zeiten eingegeben wurde

Aufgrund einer fehlerhaften Implementierung konnten keine weiteren Fälle mehr getestet werden.