Aufgabe 4: Fahrradwerkstatt

Teilnahme-ID: 66432

Team-ID: 00662

Bearbeiter/-in dieser Aufgabe:
Linus Schumann

20. November 2022

Inhaltsverzeichnis

1 Lösungsidee **2**
- 1.1 Feststellung gegebener Konstanten . . . 2
- 1.2 Aufgabenteil 1 . . . 2
  - 1.2.1 Abschließen eines Auftrages . . . 2
  - 1.2.2 Erste Methode . . . 3
  - 1.2.3 Zweite Methode . . . 3
- 1.3 Aufgabenteil 2 . . . 3
- 1.4 Aufgabenteil 3 . . . 3
  - 1.4.1 Bestimmung der Konstante n . . . 3

2 Umsetzung **3**
- 2.1 Allgemeines . . . 3
- 2.2 Implementation der allgemeinen Funktionen . . . 4
  - 2.2.1 Daten einlesen . . . 4
  - 2.2.2 Auftrag abschließen . . . 4
- 2.3 Implementation der ersten Methode . . . 4
- 2.4 Implementation der zweiten Methode . . . 4
- 2.5 Implementation der eigenen Methode . . . 4
  - 2.5.1 Auswertung der Konstante n . . . 4

3 Beispiele **4**
- 3.1 Beispiel 0 . . . 5
- 3.2 Beispiel 1 . . . 7
- 3.3 Beispiel 2 . . . 7
- 3.4 Beispiel 3 . . . 7
- 3.5 Beispiel 4 . . . 8

4 Quellcode **8**

1 Lösungsidee

1.1 Feststellung gegebener Konstanten

Als allererstes müssen die Informationen aus der Aufgabenstellung festgestellt werden. Dazu zählen in diesem Fall die Öffnungszeiten. Diese sind jeden Tag identisch und liegen bei 9 bis 17 Uhr. Da die Beispieleingaben alle auf Minuten basieren, macht es Sinn die Öffnungszeiten und alle anderen Zeiten auch in Minuten anzugeben. Daher ergeben sich folgende Konstanten:

- Dauer eines Tages (dt): 1440 min
- Dauer der offenen Phase (do): 480 min
- Dauer der geschlossenen Phase (dgo): 960 min
- Zeitpunkt der Öffnung (to): 540 min
- Zeitpunkt des Schließens (tgo): 1020 min

1.2 Aufgabenteil 1

1.2.1 Abschließen eines Auftrages

Jetzt muss überlegt werden, was beim Bearbeiten einer Aufgabe passiert, also wie sich die Gesamtzeit ändert. Dazu kann man sich folgenden Ablauf überlegen:

1. Zuerst muss die Dauer des Auftrages so angepasst werden, dass die Öffnungszeiten der Werkstatt berücksichtigt werden. Dies kann wie folgt berechnet werden:

$$newDuration = (duration/do) * dt + (duration \mod do) \tag{1}$$

   Dies bewirkt eigentlich nur, dass für jeden angebrochenen Arbeitstag, die Zeit addiert wird, die das Geschäft geschlossen ist.

2. Außerdem muss noch falls der Eingangszeitpunkt des Auftrags nach dem aktuellen Zeitpunkt liegt, die Zeit auf den Eingangszeitpunkt gesetzt werden.

$$time = startTime \tag{2}$$

3. Nachdem die Dauer nun an die Arbeitszeiten und der aktuelle Zeitpunkt an den Eingangszeitpunkt angepasst wurde, kann nun die Dauer zur Gesamtzeit addiert werden:

$$time = time + newDuration \tag{3}$$

4. Nach diesem Schritt muss festgestellt werden, ob die Endzeit innerhab der Arbeitszeiten liegt. Dazu muss zuerst die aktuelle Tageszeit berechnet werden:

$$dayTime = time \mod dt \tag{4}$$

5. Wenn für diese $dayTime$ $daytime < to$ gilt, liegt der Endzeitpunkt vor der Öffnung und die Dauer der geschlossenen Phase muss noch addiert werden:

$$time = time + dgo \tag{5}$$

6. Dannach ist nur noch möglich, dass der Endzeitpunkt (Erneute Berechnung aus Schritt 3) nach der Schließung der Werkstatt liegt. Er kann nicht mehr am nächsten Tag liegen. Daher muss wenn $dayTime > tgo$ gilt, erneut die Dauer der geschlossenen Phase addiert werden:

$$time = time + dgo \tag{6}$$

7. Am Ende ist es dann möglich die Wartezeit zu berechnen:

$$waitingTime = time - startTime \tag{7}$$

1.2.2 Erste Methode

Die erste Methode ist sehr simpel, da einfach nur die Aufträge in der Reihenfolge des Einganges bearbeitet werden. Dafür wird einfach für jeden Auftrag, der in Abschnitt 1.2.1 Ablauf ausgeführt und die resultierende Wartezeit gespeichert. Daraus kann dann die maximale und durchschnittliche Wartezeit berechnet werden.
Die Reihenfolge der Bearbeitung spiegelt das Verhalten einer normalen Queue wieder.

1.2.3 Zweite Methode

Die zweite Methode ist im Vergleich zur ersten Methode schon etwas komplexer, da nicht die Reihenfolge des Einganges eingehalten wird, sondern immer der kürzeste Auftrag zuerst bearbeitet wird.
Daher entspricht diese Reihenfolge das Konzept einer Priority Queue.

1.3 Aufgabenteil 2

Das Problem der zweiten Methode (Abschnitt 1.2.3) liegt darin, dass eine sehr hohe Maximale Wartezeit entsteht. Dies wird auch durch die Auswertung des ersten Beispiel (Abschnitt 3.1) klar deutlich.
Die hohe Wartezeit wird natürlich dadurch erreicht, dass lange Aufträge bei der zweiten Methode sehr oft nach hinten verschoben werden. Allerdings wird dadurch eine sehr niedrige durchschnittliche Wartezeit erreicht.

1.4 Aufgabenteil 3

Nun muss eine neue Methode erarbeitet werden. Dabei macht es Sinn sich mit den Problemen der 2. Methode auseinander zu setzen. Wie in Abschnitt 1.3 beschrieben wurde, entsteht die hohe maximale Wartezeit dadurch, dass lange Aufträge sehr oft verschoben werden.
Um dies zum Teil zu verhindert, macht es Sinn eine maximale Anzahl an Verschiebungen nach hinten zu definieren. Diese Maximale Anzahl wird nun mit der Formel $|Warteschlange| * n$ berechnet. n ist eine Konstante für die im Folgenden ein geeigeneter Wert gefunden wird.

1.4.1 Bestimmung der Konstante n

Bei der Bestimmung der Konstante n muss zuerst ein Zusammenhang zwischen der maximalen und durchschnittlichen Wartezeit hergestellt werden. Im Folgenden wird dazu eine "Punktzahl" mit dieser Formel berechnet:

$$score = \max Wartezeiten + \overline{Wartezeiten} \tag{8}$$

In der Menge $Wartezeiten$ ist für jeden Auftrag die Wartezeit gespeichert.
Dann kann für $0 \leq n \leq 100$ jeweils die Methode ausgeführt werden und die maximale Punktzahl mit dem zugehörigen Wert für n gespeichert werden.
Wenn dies für jede Beispieldatei ausgeführt wird, kann ein Gesamtdurchschnitt gebildet werden, der fest ins Programm eingebunden werden kann. Eine genauere Auswertung lässt sich in Abschnitt 2.5.1 finden.

2 Umsetzung

2.1 Allgemeines

Im Folgenden wird die Umsetzung der in Abschnitt 1 beschriebene Lösungsidee, näher erläutert. Grundsätzlich wurde diese Idee dabei in C++, genauer gesagt in der Datei "Aufgabe_4.cpp" implementiert. Diese Datei befindet sich im Verzeichnis "./source/".

Um das implementierte Programm zu starten kann das Batch Skript "Aufgabe_4.bat" im Verzeichnis "./executables/" genutzt werden. Dieses startet das von mir für Windows kompilierte Programm ("Aufgabe_4.exe"). Für andere Betriebssysteme müsste die Source-Datei erneut auf dem entsprechenden Rechner kompiliert werden.

Im Verzeichnis "./beispieleingaben/" befinden sich alle in dieser Dokumentation aufgeführten Beispiele und im Verzeichnis "./beispielausgaben/" befinden sich dementsprechend die gesicherten Ausgaben

Letztere werden mit der Datei-Endung ".out" gespeichert. Außerdem werden bei jedem Beispiel drei csv-Dateien gesichert, für jede Methode eine. In diesen wird jeweils für jeden Auftrag die Dauer in Verbindung mit der Wartezeit angegeben. Eine Übersicht dieser Daten ist in der Excel-Datei "Analyse.xlsx" zu finden. Dabei wurde für jede csv-Datei ein Tabellenblatt mit einem Diagramm erstellt.

2.2 Implementation der allgemeinen Funktionen

2.2.1 Daten einlesen

Das Einlesen der Daten ist relativ simpel, da einfach die Eingabedatei Zeile für Zeile ausgelesen wird. Die zwei Zahlen die sich dabei in einer Zeile befinden werden dann einfach als "Pair" in einer Queue gespeichert. Es macht Sinn eine Queue zu nutzen, da man immer nur den nächsten Auftrag auslesen kann und muss.
Das Einlesen wurde in der Funktion "readData" implementiert.

2.2.2 Auftrag abschließen

Beim Abschließen eines Auftrages können einfach die mathematischen Schritte aus Abschnitt 1.2.1 verwendet werden. Zusätzlichen wird am Schluss die Wartezeit einem Vektor angefügt und geprüft, ob ein neues Maximum verliegt.
Das Abschließen eines Auftrages wurde in "finishJobïmplementiert.

2.3 Implementation der ersten Methode

Die Implementation der ersten Methode geht einfach nur die Eingabedaten durch. Dabei wird immer der erste Auftrag der Queue bearbeitet und dann gelöscht. Dies passiert solange, bis die Queue leer ist.

2.4 Implementation der zweiten Methode

Die zweite Methode nutzt eine Priority Queue, um alle aktuell verliegenden Aufträge zu speichern. Diese Priority Queue speichert die Aufträge aufsteigend, sodass am Anfang der Queue immer der kürzeste Auftrag vorliegt. Um die "Pair-Werte", die im Input in diesem Format gespeichert wurden: "Zeit, Dauer", in der Priority Queue nach der Dauer zu sortieren, muss die Reihenfolge innerhalb des Pairs getauscht werden.
Der Ablauf sieht insgesamt so aus, dass solange Eingabedaten verfügar sind, alle bis zum aktuellen Zeitpunkt vorliegenden Aufträge eingelesen werden und dann der kürzeste von diesen bearbeitet wird. Falls zu einem Zeitpunkt noch kein Auftrag vorliegt, wird einfach der nächste gewählt.

2.5 Implementation der eigenen Methode

Die eigene Methode funktioniert ähnlich wie die zweite Methode. Es wird nur für jeden Auftrag in der "Priority Queue" auch noch die Anzahl an Verschiebungen gespeichert. Beim jeden Abschließen eines Auftrages wird diese Anzahl für jedes Element um 1 erhöht. Um dies umzusetzen kann allerdings keine richtige Priority Queue verwedet werden, da es meines Wissens nicht einfach so möglich ist, über eine Queue zu iterieren. Allerdings kann die Funktionsweise einer Priority Queue auch mit einem Vektor umgesetzt werden, der nach jedem Hinzufügen eines Elementes in den Vektor, sortiert wird. Die Funktion zum Sortieren wurde selbst implementiert und sortiert eigentlich alle Elemente aufsteigend. Allerdings werden Elemente deren Anzahl an Verschiebungen höher ist als die maximale Verschiebung, an den Anfang des Vektors "geschoben".

2.5.1 Auswertung der Konstante n

Wenn man, dass in Abschnitt 1.4.1 Verfahren anwendet, kommt man insgsamt auf einen Wert von $n = 3$. Dieser lässt sich auf jedes Beispiel anwenden und verbessert auch jedes Beispiel im Vergleich zur ersten bzw. zweiten Methode deutlich.

3 Beispiele

In diesem Abschnitt befinden sich alle Ausgaben zu allen Beispielen. Außerdem werden zum Beispiel 0 auch noch 3 Diagramme zur Auswertung angegeben.

3.1 Beispiel 0

Das ”erste” Beispiel wird nun besonders detailiert ausgewertet:
Insgesamt kann man auf Grundlage der Diagramme für die maximale und durchschnittliche Wartezeit feststellen, dass bei der ersten Methode zwar die maximale Wartezeit sehr gering ist, allerdings die durchschnittliche Wartezeit vergleichsweise sehr hoch ist. Die zweite Methode verbessert die durchschnittliche Wartezeit dann enorm, darunter leidet allerdings die maximale Wartezeit deutlich. Schlussendlich verbessert die eigene Methode die durschnittliche Wartezeit im Vergleich zur ersten Methode und die maximale Wartezeit im Vergleich zur zweiten. Daher kann man sagen, dass die eigene Methode beide vorherigen Methoden ”kombiniert” und damit die beste Methode ist und meinen Erwartungen somit entspricht.

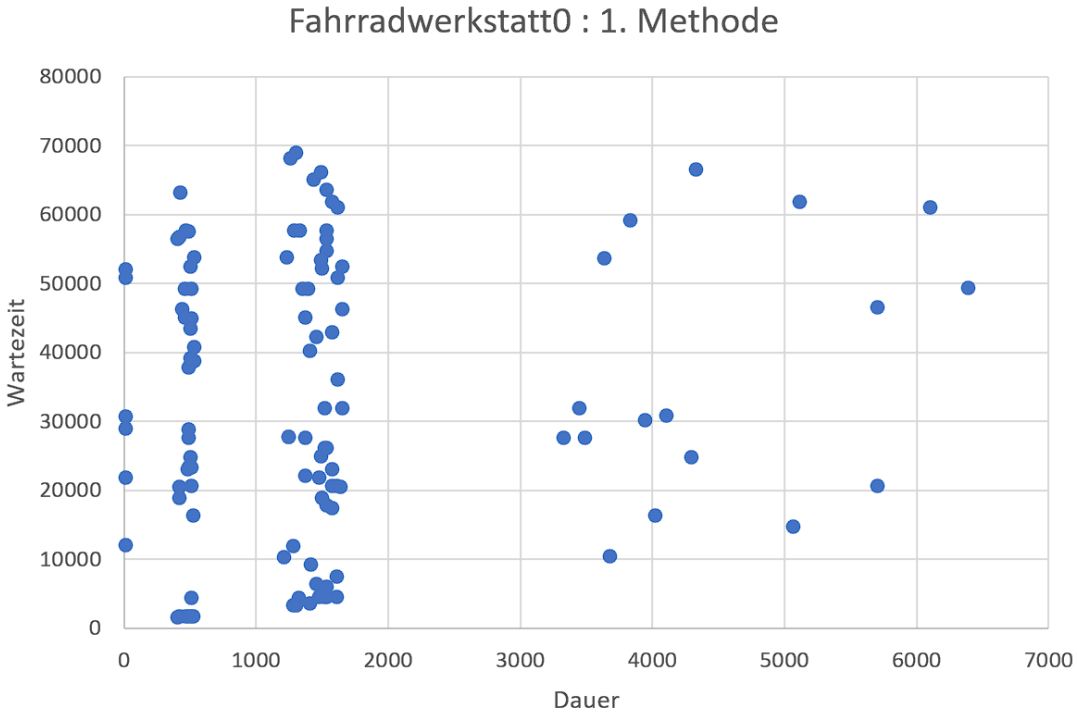

Abbildung 1: Auswertung Beispiel 0: Erste Methode

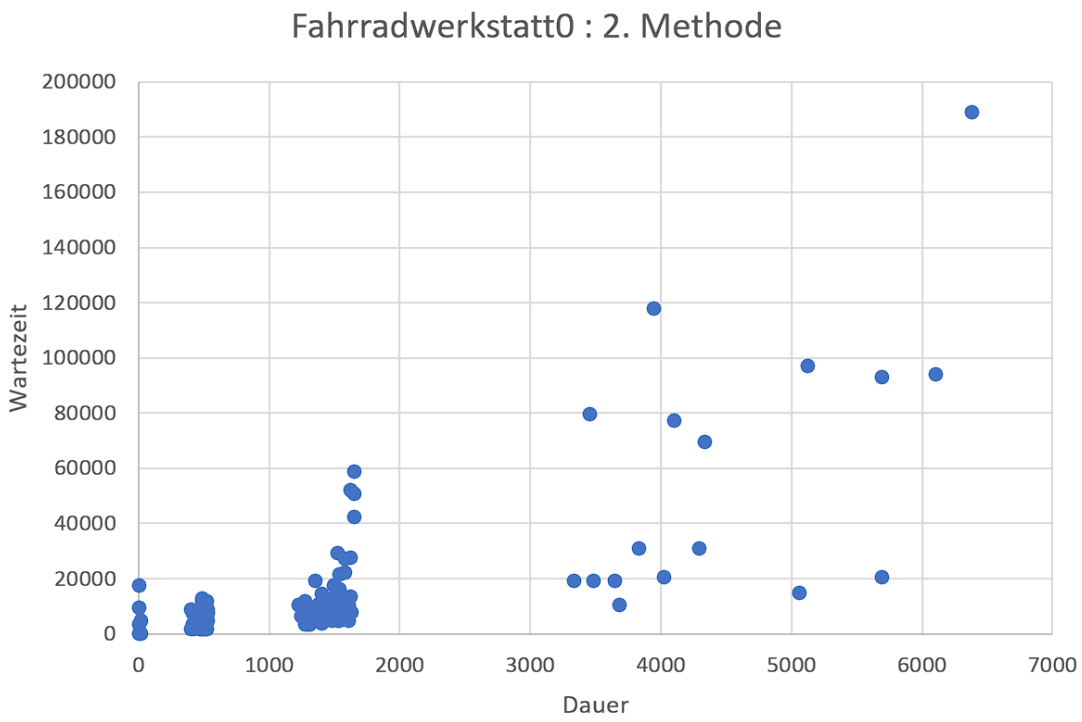

Abbildung 2: Auswertung Beispiel 0: Zweite Methode

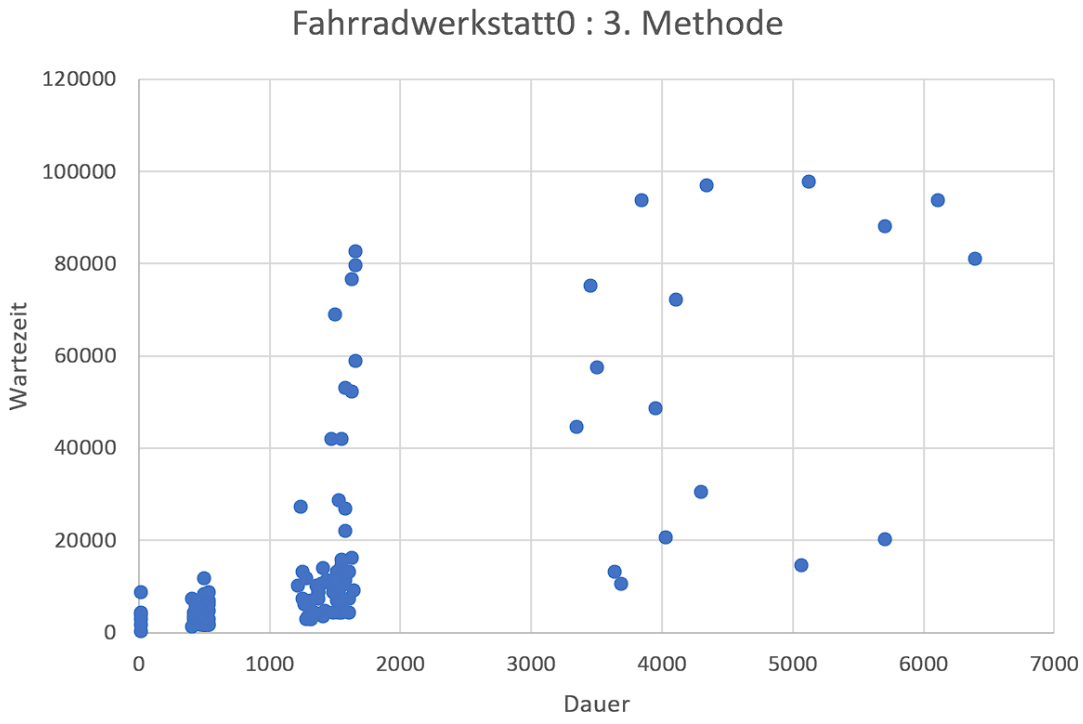

Abbildung 3: Auswertung Beispiel 0: Eigene Methode

```
Erste Methode von Mark
Durchschnittliche Wartezeit: 22 d, 17 h, 53 m,
```

```
Maximale Wartezeit: 47 d, 18 h, 11 m,

Zweite Methode von Mark
Durchschnittliche Wartezeit: 11 d, 19 h, 1 m,
Maximale Wartezeit: 131 d, 1 h, 34 m,

Eigene Methode
Durchschnittliche Wartezeit: 13 d, 4 h, 22 m,
Maximale Wartezeit: 67 d, 20 h, 1 m,
```

Listing 1: Ausgabe Beispiel 0

3.2 Beispiel 1

```
Erste Methode von Mark
Durchschnittliche Wartezeit: 44 d, 2 h, 55 m,
Maximale Wartezeit: 89 d, 2 h, 41 m,

Zweite Methode von Mark
Durchschnittliche Wartezeit: 8 d, 6 h, 3 m,
Maximale Wartezeit: 301 d, 2 h, 3 m,

Eigene Methode
Durchschnittliche Wartezeit: 20 d, 7 h, 35 m,
Maximale Wartezeit: 113 d, 4 h, 57 m,
```

Listing 2: Ausgabe Beispiel 1

3.3 Beispiel 2

```
Erste Methode von Mark
Durchschnittliche Wartezeit: 35 d, 13 h, 14 m,
Maximale Wartezeit: 77 d, 1 h, 33 m,

Zweite Methode von Mark
Durchschnittliche Wartezeit: 10 d, 6 h, 53 m,
Maximale Wartezeit: 227 d, 3 h, 27 m,

Eigene Methode
Durchschnittliche Wartezeit: 15 d, 1 h, 44 m,
Maximale Wartezeit: 84 d, 3 h, 57 m,
```

Listing 3: Ausgabe Beispiel 2

3.4 Beispiel 3

```
Erste Methode von Mark
Durchschnittliche Wartezeit: 20 d, 20 h, 28 m,
Maximale Wartezeit: 42 d, 5 h, 41 m,

Zweite Methode von Mark
Durchschnittliche Wartezeit: 11 d, 23 h, 22 m,
Maximale Wartezeit: 265 d, 6 h, 56 m,

Eigene Methode
Durchschnittliche Wartezeit: 12 d, 17 h, 17 m,
Maximale Wartezeit: 80 d, 4 h, 27 m,
```

Listing 4: Ausgabe Beispiel 3

3.5 Beispiel 4

```
Erste Methode von Mark
Durchschnittliche Wartezeit: 51 d, 16 h, 27 m,
Maximale Wartezeit: 116 d, 0 h, 19 m,

Zweite Methode von Mark
Durchschnittliche Wartezeit: 29 d, 7 h, 20 m,
Maximale Wartezeit: 252 d, 4 h, 35 m,

Eigene Methode
Durchschnittliche Wartezeit: 30 d, 17 h, 37 m,
Maximale Wartezeit: 146 d, 1 h, 3 m,
```

Listing 5: Ausgabe Beispiel 4

4 Quellcode

```cpp
#include <bits/stdc++.h>

#define pii pair<int,int>
#define tiii tuple<int,int,int>
#define ve vector
#define INF_INT numeric_limits<int>::max()/2

using namespace std;

ifstream inputFile; // input file of example
ofstream outputFile; // output file for text
ofstream outputFileStatistic1; // statistic files
ofstream outputFileStatistic2;
ofstream outputFileStatistic3;

const int dayDuration = 1440; // duration of one full day
const int openDuration = 480; // duration the shop is opened
const int closeDuration = 960; // duration the shop is closed
const int openTime = 540; // time when shop opens
const int closeTime = 1020; // time when shop closes

int n = 3; // best n for all examples is 3
int bestN; // save best n when calculating n
int maxBackMoves; // save max times job can be moved back
int maxScore = INF_INT; // save score to calculate best n

void finishJob(int startTime, int duration, int &time, ve<pii> &waitingTimes, int &
    ↪ maxWaitingTime){
    int durationSave = duration;
    duration = ((duration / openDuration)*dayDuration) + (duration % openDuration);
    if(startTime > time) time = startTime;
    time += duration;
    int dayTime = time%dayDuration;
    if(dayTime < openTime){
        time += closeDuration;
        dayTime = time%dayDuration;
    }
    if(dayTime > closeTime) time += closeDuration;
    int waitingTime = time - startTime;
    assert((time%dayDuration) >= openTime && (time%dayDuration) <= closeTime);
    waitingTimes.emplace_back(waitingTime, durationSave);
    maxWaitingTime = max(maxWaitingTime, waitingTime);
}

void printTime(int time){
    cout << (time/dayDuration) << "␣d,␣" << ((time%dayDuration)/60) << "␣h,␣" << ((time%
    ↪ dayDuration)%60) << "␣m,␣" << endl;
    outputFile << (time/dayDuration) << "␣d,␣" << ((time%dayDuration)/60) << "␣h,␣" << ((
    ↪ time%dayDuration)%60) << "␣m,␣" << endl;
```

```
}

void calculateAndPrintResults(ve<pii> &waitingTimes, int maxWaitingTime, ofstream &
    ↪ outputFileStatistik, bool print){
    int averageWaitingTime = 0;
    for(auto x : waitingTimes){
        averageWaitingTime += x.first;
        if(print) outputFileStatistik << x.second << "," << x.first << endl;
    }
    averageWaitingTime  = averageWaitingTime / waitingTimes.size();
    int score = maxWaitingTime+(averageWaitingTime*5);
    if(maxScore > score){
        maxScore = score;
        bestN = n;
    }
    if(print){
        cout << "Score: " << score << endl;
        cout << "Durchschnittliche Wartezeit: ";
        outputFile << "Durchschnittliche Wartezeit: ";
        printTime(averageWaitingTime);
        cout << "Maximale Wartezeit: ";
        outputFile << "Maximale Wartezeit: ";
        printTime(maxWaitingTime);
    }
}

void firstSolution(queue<pii> data, bool print){
    int time = 0, maxWaitingTime = 0, startTime, duration;
    ve<pii> waitingTimes;
    while(!data.empty()){
        tie(startTime, duration) = data.front();
        data.pop();
        finishJob(startTime, duration, time, waitingTimes, maxWaitingTime);
    }
    calculateAndPrintResults(waitingTimes, maxWaitingTime, outputFileStatistic1, print);
}

void secondSolution(queue<pii> data, bool print){
    int time = 0, maxWaitingTime = 0, nextStartTime = 0, duration, startTime;
    ve<pii> waitingTimes;
    priority_queue<pii, ve<pii>, greater<pii>> pq;
    while(!data.empty() || (data.empty() && !pq.empty())){
        while((nextStartTime <= time || pq.empty()) && !data.empty()){
            tie(startTime, duration) = data.front();
            pq.push(make_pair(duration, startTime));
            data.pop();
            if(!data.empty()) nextStartTime = data.front().first;
        }
        tie(duration, startTime) = pq.top();
        pq.pop();
        finishJob(startTime, duration, time, waitingTimes, maxWaitingTime);
    }
    calculateAndPrintResults(waitingTimes, maxWaitingTime, outputFileStatistic2, print);
}

bool thirdSortMethod(tiii x1, tiii x2){
    if(get<2>(x1) > maxBackMoves && get<2>(x2) > maxBackMoves) return get<2>(x1) > get
    ↪ <2>(x2);
    if(get<2>(x1) > maxBackMoves) return true;
    if(get<2>(x2) > maxBackMoves) return false;
    return x1 < x2;
}

void thirdSolution(queue<pii> data, bool print){
    int time = 0, maxWaitingTime = 0, nextStartTime = 0, duration, startTime, skipped;
    ve<pii> waitingTimes;
    ve<tiii> pq;
    while(!data.empty() || (data.empty() && !pq.empty())){
        while((nextStartTime <= time || pq.empty()) && !data.empty()){
            tie(startTime, duration) = data.front();
            pq.emplace_back(duration, startTime, 0);
            data.pop();
            if(!data.empty()) nextStartTime = data.front().first;
```

```
        }
        maxBackMoves = pq.size()*n;
        sort(pq.begin(), pq.end(), thirdSortMethod);
        tie(duration, startTime, skipped) = pq.front();
        pq.erase(pq.begin());
        finishJob(startTime, duration, time, waitingTimes, maxWaitingTime);
        for(int i = 0; i < pq.size(); i++){
            get<2>(pq[i])++;
        }
    }
    calculateAndPrintResults(waitingTimes, maxWaitingTime, outputFileStatistic3, print);
}

void readData(queue<pii> &data){
    string line;
    while(getline(inputFile, line)){
        if(line != "␣"){
            string startTimeString = (line.substr(0, line.find("␣")));
            line.erase(0, line.find("␣") + 1);
            string durationString = line;
            int startTime = stoi(startTimeString);
            int duration = stoi(durationString);
            data.push(make_pair(startTime, duration));
        }
    }
}

void solve(bool testN){
    queue<pii> data;
    readData(data);
    if(testN){
        for(n = 0; n < 1000; n++){
            thirdSolution(data, false);
        }
        cout << "Best␣N␣is:␣" << bestN << endl;
    } else {
        cout << endl << "Erste␣Methode␣von␣Mark" << endl;
        outputFile << "Erste␣Methode␣von␣Mark" << endl;
        firstSolution(data, true);
        cout << endl << "Zweite␣Methode␣von␣Mark" << endl;
        outputFile << endl << "Zweite␣Methode␣von␣Mark" << endl;
        secondSolution(data, true);
        cout << endl << "Eigene␣Methode␣mit␣n␣=␣" << n << endl;
        outputFile << endl << "Eigene␣Methode" << endl;
        thirdSolution(data, true);
        cout << endl;
    }
}

int main(){
    string filename, istest;
    bool testN;
    cout << "Enter␣number␣of␣example:␣";
    cin >> filename;
    cout << "Calculate␣best␣n␣for␣example␣or␣normal␣run??␣(0/1)" << endl;
    cout << "->␣";
    cin >> istest;
    testN = (istest == "0");
    inputFile.open("../beispieleingaben/fahrradwerkstatt"+filename+".txt");
    if(!inputFile.is_open()){
        cout << "Unable␣to␣open␣file" << endl;
        return 0;
    } else {
        if(!testN){
            outputFile.open("../beispielausgaben/fahrradwerkstatt"+filename+".out");
            outputFileStatistic1.open("../beispielausgaben/fahrradwerkstatt"+filename+"
↪ _sta1.csv");
            outputFileStatistic2.open("../beispielausgaben/fahrradwerkstatt"+filename+"
↪ _sta2.csv");
            outputFileStatistic3.open("../beispielausgaben/fahrradwerkstatt"+filename+"
↪ _sta3.csv");
        }
    }
```

```
    solve(testN);

    inputFile.close();
    if(!testN){
        outputFile.close();
        outputFileStatistic1.close();
        outputFileStatistic2.close();
    }
    return 0;
}
```