Aufgabe 1: Wundertüte

Team-ID: 00129

Team-Name: VGxSRlowMTZTV2RPYW...

Bearbeiter/-innen dieser Aufgabe:
Felix Mölle

19. November 2023

Inhaltsverzeichnis

Kompletter Teamname:
VGxSRlowMTZTV2RPYWxGblRrUlpaMDVVVVdkT1JGVm5Ua2RGWjA1dFVXZE9ha2xuVFhwQlow
MTZWV2RPUkVsblRsUlZaMDVVVldkT1ZGbG5Ua2RSWjA1VVRXZE9SR05uVG1wUlowNTZXV2RPV
kVWblRsUkZaMDB5VVdkTk1sRTk=

1 Lösungsidee

Die zentrale Herausforderung besteht darin, die Spiele möglichst gleichmäßig auf die Wundertüten zu verteilen und dabei die Unterschiede in der Anzahl der Spiele zwischen den Tüten zu minimieren.

Der Verteilalgorithmus arbeitet iterativ, um sicherzustellen, dass die Spiele nach Möglichkeit gleichmäßig auf die Tüten aufgeteilt werden.

Zunächst wird für jede Spieleart berechnet, wie viele Spiele theoretisch auf jede Wundertüte kommen sollten, wenn die Verteilung vollständig gleichmäßig wäre. Dabei wird der Divisionsoperator verwendet, um eine erste Schätzung zu erhalten.

Der Rest dieser Division gibt an, wie viele Spiele auf ungleich verteilte Tüten kommen müssen.
Um die ungleich verteilten Spiele zu berücksichtigen, wird für jede verbleibende Spieleart iterativ durch die Wundertüten iteriert.
Dabei wird die Spieleart der aktuellen Wundertüte mit den ungleich verteilten Spielen aufgefüllt.
Dies geschieht, indem der Index auf die aktuelle Tüte und die Spieleart verwendet wird.
Der Index wird dabei nach jedem Hinzufügen inkrementiert und bei Erreichen der letzten Tüte auf null zurückgesetzt.
Dies gewährleistet eine gleichmäßige Verteilung der ungleich verteilten Spiele auf die Tüten.

Durch diese iterative Verteilung wird erreicht, dass die Spiele möglichst gleichmäßig auf die Wundertüten aufgeteilt werden, und die Unterschiede in der Anzahl der Spiele zwischen den Tüten minimal sind.
Die Verwendung von Schleifen und mathematischen Operationen ermöglicht eine effiziente Implementierung dieses Verteilalgorithmus in Rust.

2 Umsetzung

Die Implementierung erfolgt in Rust, einer Programmiersprache, die für ihre Sicherheit, Geschwindigkeit und Nebenläufigkeit bekannt ist.

2.1 Einlesen der Eingabedaten

Die Funktion
`read_data(fixed_path: String) -> (i32, i32, Vec<i32>)`
ist verantwortlich für das Einlesen der Eingabedaten.
Sie erwartet entweder den Pfad zu einer Datei als Argument oder fordert den Benutzer zur Eingabe auf.
Der Dateiinhalt wird mit Hilfe der `BufReader`- und `lines()`-Methoden verarbeitet.
Die Anzahl der Spiele jeder Sorte sowie die Gesamtanzahl der Spiele werden anschließend zurückgegeben.

2.2 Initialisierung der Wundertüten

Die Funktion
`init_bags(bag_count: i32) -> Vec<String>`
initialisiert leere Wundertüten als Vektoren von Zeichenketten.
Jede Wundertüte wird dabei als leere Zeichenkette repräsentiert.

2.3 Verteilungsalgorithmus

Die Spiele werden gemäß den Vorgaben gleichmäßig auf die Wundertüten verteilt.
In der `main()`-Funktion wird für jedes Spiel die Anzahl berechnet, die theoretisch auf jede Tüte kommen sollte.
Diese Spiele werden dann gleichmäßig auf die Tüten verteilt, und der Rest wird auf ungleich verteilte Tüten aufgeteilt.

Die Funktion
`append_asym(index: usize, game: usize, bag_count: i32, pbags: &mut Vec<String>) -> usize`
spielt eine entscheidende Rolle bei der Verteilung der ungleich verteilten Spiele.
Sie fügt der aktuellen Wundertüte das Spiel der aktuellen Sorte hinzu und stellt sicher, dass die ungleich verteilten Spiele möglichst gleichmäßig auf die Tüten verteilt werden.
Der Index wird verwendet, um die Tüten nacheinander zu durchlaufen.

Die Ergebnisse, bestehend aus den gefüllten Wundertüten, werden am Ende ausgegeben.

Diese Implementierung ermöglicht eine effiziente und korrekte Umsetzung des Verteilalgorithmus in Rust.
Der Fokus liegt darauf, die Spiele möglichst gleichmäßig zu verteilen und
gleichzeitig die Anzahlunterschiede zwischen den Wundertüten zu minimieren.

3 Beispiele

3.1 Wundertüte 1

```
bags: 6; with 3 games. Games: [18, 6, 12]
["000122", "000122", "000122", "000122", "000122", "000122"]
```

3.2 Wundertüte 2

```
bags: 9; with 4 games. Games: [10, 9, 3, 5]
["001", "012", "012", "012", "013", "013", "013", "013", "013"]
```

3.3 Wundertüte 3

```
bags: 11; with 5 games. Games: [2, 11, 6, 2, 1]
["01", "01", "12", "12", "12", "12", "12", "12", "13", "13", "14"]
```

3.4 Wundertüte 4

```
bags: 17; with 6 games. Games: [21, 114, 38, 97, 513, 89]
[
"00111111222333334444444444444444444444444444444455555",
"00111111222333334444444444444444444444444444444555555",
"00111111222333334444444444444444444444444444455555555",
"00111111223333334444444444444444444444444444455555555",
"01111111223333334444444444444444444444444444455555555",
"01111111223333334444444444444444444444444444455555555",
"01111111223333334444444444444444444444444444455555555",
"01111111223333334444444444444444444444444444455555555",
"01111111223333334444444444444444444444444444455555555",
"01111111223333334444444444444444444444444444455555555",
"01111111223333334444444444444444444444444444455555555",
"01111111223333334444444444444444444444444444455555555",
"01111111223333334444444444444444444444444444455555555",
"01111111223333334444444444444444444444444444455555555",
"01111111223333334444444444444444444444444444455555555",
"01111111223333334444444444444444444444444444455555555",
"01111111223333334444444444444444444444444444455555555"
]
```

3.5 Wundertüte 5

```
bags: 97; with 23 games. Games: [
50, 41, 163, 124, 113, 83, 129, 65, 70, 56, 127, 83,
24, 69, 182, 174, 76, 106, 92, 155, 29, 133, 121 ]

[
    "022344668910111314141515171819192121 22",
    ...
    "022345668910111314141516171819192122 22",
    "022345668101011131415151617181919212222",
    ...
    "223446689101113141415151718191921212 2",
    "223446689101113141415151718191921212 2",
    "223446689101113141415151718191921212 2"
]
```

4 Quellcode

4.1 main.rs: Hauptprogramm

```rust
use std::env;
use std::io::{BufRead, BufReader};

fn main() {
    let args: Vec<String> = env::args().collect();
    let mut fixed_path = String::new();

    if args.len() >= 2 {
        fixed_path = args[1].clone();
    }

    let (bag_count, game_count, games) = read_data(fixed_path);

    println!(
        "bags: {}; with {} games. Games: {:?}",
        bag_count, game_count, games
    );

    let mut bags = init_bags(bag_count);

    let mut index: usize = 0;
    for (i, game) in games.iter().enumerate() {
        let split = game / bag_count;
        for _ in 0..split {
            for bag in &mut bags {
                bag.push_str(&i.to_string());
            }
        }

        for _ in 0..(game - (split * bag_count)) {
            let pbags: &mut Vec<String> = &mut bags;
            index = append_asym(index, i, bag_count, pbags);
        }
    }

    println!("{:?}", bags);
}

fn read_data(fixed_path: String) -> (i32, i32, Vec<i32>) {
    let mut input = String::new();

    if fixed_path.is_empty() {
        println!("Enter path to file:");
        std::io::stdin().read_line(&mut input).unwrap();
    } else {
        input = fixed_path;
    }

    let file_path = input.trim();
    let file = BufReader::new(std::fs::File::open(file_path).unwrap());

    let mut lines = file.lines();

    let bag_count: i32 = lines.next().unwrap().unwrap().parse::<i32>().unwrap();
```

```
55          let game_count: i32 = lines.next().unwrap().unwrap().parse::<i32>().unwrap();

57          let mut games: Vec<i32> = Vec::new();

59          for line in lines {
                games.push(line.unwrap().parse::<i32>().unwrap());
61          }

63          (bag_count, game_count, games)
        }
65
        fn init_bags(bag_count: i32) -> Vec<String> {
67          let mut bags: Vec<String> = Vec::new();
            for _ in 0..bag_count {
69              bags.push(String::new());
            }
71          bags
        }
73
        fn append_asym(index: usize, game: usize, bag_count: i32, pbags: &mut Vec<String>) -> usize {
75          (*pbags)[index].push_str(&game.to_string());
            if index == (bag_count - 1) as usize {
77              0
            } else {
79              index + 1
            }
81      }
```