Aufgabe 3: Wandertag

Team-ID: **00078**

Team: Timon

Bearbeiter dieser Aufgabe:
Timon Retzlaff

Inhaltsverzeichnis

Lösungsidee

Ich iteriere über alle Möglichkeiten für die Verteilung der drei Streckenlängen auf die Minima der Personen und zähle, wie viele Personen jeweils teilnehmen. Währenddessen merke ich mir die Variante mit den meisten Personen. Falls bei einer möglichen Verteilung klar ist, dass sie das Ergebnis nicht verbessern kann, überspringe ich sie. Am Ende gebe ich die beste Lösung aus. Dadurch hat das Programm $O(n^4)$. Ich habe auch eine effizientere Lösung mit $O(n^3)$ gefunden, diese jedoch nicht umgesetzt, da, durch eine stark optimierte Personenzählung, selbst das

anspruchsvollste Beispiel innerhalb von ~400ms berechnet werden konnte und mir eine Optimierung daher unnötig erschien.

Umsetzung

Einlesen der Datei

Die Implementierung beginnt damit, dass die Eingabedaten aus einer Datei gelesen und in zwei Arrays (*values* und *persons*) gespeichert werden. Das Array *values* enthält die minimalen Streckenlängen der Personen, während *persons* Objekte der Klasse *Person* mit den jeweiligen Mindest- und Maximalwerten speichert. Die Methode *initializeArrays* übernimmt das Parsen der Daten.

Entfernen von Duplikaten und Sortieren

Das Array *values* wird von doppelten Werten bereinigt, indem die Werte in ein *HashSet* überführt werden. Die Anzahl der entfernten Duplikate wird in *doubledCount* gespeichert, was später genutzt wird, um Berechnungen zu optimieren. Anschließend werden die Werte in *values* sortiert.

BitSet-Initialisierung

Die Methode *initializeBitSets* erstellt für jede Streckenlänge einen *MyBitSet*, in dem das Bit an der Position *i* gesetzt wird, wenn diese Streckenlänge in den Bereich [*min*, *max*] der Person *i* fällt.

Die *MyBitSet*-Klasse unterstützt effiziente Mengenoperationen durch Bitmanipulation auf long-Arrays und enthält Methoden wie *set*, um ein bestimmtes Bit zu aktivieren, und *orCardinality*, um die Anzahl der gesetzten Bits zu zählen und vorher den Oder-Operator auf den aktuellen Wert und ein anderes *MyBitSet* anzuwenden. Die Kombination der beiden Methoden beschleunigt den Code an dieser Stelle um etwa Faktor 2.

Iterationen und Kombinationsberechnung

In den verschachtelten Schleifen wird über mögliche Kombinationen von drei Streckenlängen (*i*, *j*, *k*) iteriert. Für jede Kombination wird die Gesamtzahl der abgedeckten Personen mit der Methode *orCardinality* berechnet, die die Anzahl der Bits zählt, die in einer *OR-Verknüpfung* der drei *MyBitSets* gesetzt sind. Hierbei wird der Wert für die Bitmengenoperationen durch die Methode *copyOr* vorbereitet, die die zwei *BitSets* der äußeren Schleifen (*i*,*j*) kombiniert und das Ergebnis im *preComputed*-BitSet speichert. Durch das Wiederverwenden des vorberechneten Bit-Sets wird das Programm erneut ca. um Faktor 2 schneller.

Futility Pruning

Um unnötige Berechnungen zu vermeiden, wird in der innersten Schleife(*k*) in jeder Iteration überprüft, ob die maximale Anzahl an Teilnehmern, die noch durch weitere Streckenlängen hinzukommen könnten, das bisher beste Ergebnis übertreffen könnte. Wenn dies nicht der Fall ist, wird die aktuelle Schleifeniteration (*i*,*j*) vorzeitig beendet.

Das wird erreicht, indem man berechnet wie viele Minima noch hinter dem aktuellen index *k* liegen (maximale zusätzliche Teilnehmer). Falls dies plus *doubledCount*(s.o.) nicht die bisher beste Lösung schlägt, kann die innerste Schleife abgebrochen werden. Diese Schätzung ist immer zu hoch oder richtig, wodurch gewährleistet wird, dass die beste Lösung immer gefunden wird.

Ergebnisdarstellung

Nachdem die beste Kombination gefunden wurde, werden die zugehörigen Streckenlängen und die Anzahl der abgedeckten Personen ausgegeben. Die Gesamtzeit der Berechnung wird gemessen und in Millisekunden angezeigt.

Beispiele

Beispiel 1: Standardfall

Stellen wir uns einen typischen Eingabefall vor: Es gibt mehrere Personen mit minimalen und maximalen Streckenlängen. Das Programm durchläuft alle möglichen Kombinationen von drei Streckenlängen und gibt diejenige Kombination aus, die die maximale Anzahl an Personen abdeckt. Ein Beispiel:

Eingabe:

- 5
- 3 10
- 2 8
- 5 12
- 1 7
- 4 11

Ausgabe:

5 people attend for the lengths: 1m; 4m; 5m;

Found in 0.163775ms

Beispiel 2: Spezialfall – Wenige Personen

Ein Spezialfall ist, wenn weniger als drei Personen gegeben sind. Dann wird direkt eine Lösung bestehend aus den Minima ausgegeben.

Eingabe:

- 2
- 1 5
- 3 7

Ausgabe:

2 people attend for the lengths: 1m; 3m

Beispiel 3: Spezialfall – Identische Minima

Ein weiterer interessanter Fall ist, wenn mehrere Personen denselben minimalen Wert haben. Wenn die Anzahl der Minima durch das Wegstreichen der doppelten Werte unter drei fällt, wird wieder eine einfache Lösung ausgegeben.

Eingabe:

- 4
- 3 9
- 3 10
- 3 8
- 3 7

Ausgabe:

4 people attend for the lengths: 3m

Beispiel 4: Spezialfall – Identische Minima

Wenn die Anzahl der Minima durch das Wegstreichen der doppelten Werte über drei bleibt, wird normal gelöst.

Eingabe:

- 6
- 3 9
- 3 10
- 3 8
- 4 7
- 5 11
- 6 13

Ausgabe:

6 people attend for the lengths: 3m; 5m; 6m

Found in 0.14953ms

Beispiel 5: Spezialfall – Große Eingabe

Zur Bewertung der Effizienz des Programms kann ein Testfall mit vielen Personen verwendet werden. An diesem Beispiel erkennt man, dass die Laufzeit des Programms selbst beim größten gegebenen Beispiel niedrig bleibt.

Eingabe:

- Beispiel 7

Ausgabe:

551 people attend for the lengths: 39520m; 76088m; 91584m

Found in 363.880645ms

Beispiel 6

Eingabe:

- Beispiel 6

Ausgabe:

330 people attend for the lengths: 42834m; 74810m; 92920m

Found in 71.30841ms

Beispiel 7

Eingabe:

- Beispiel 5

Ausgabe:

153 people attend for the lengths: 36696m; 60828m; 88584m

Found in 5.084265ms

Beispiel 8

Eingabe:

- Beispiel 4

Ausgabe:

79 people attend for the lengths: 524m; 811m; 922m

Found in 1.26366ms

Beispiel 9

Eingabe:

- Beispiel 3

Ausgabe:

10 people attend for the lengths: 19m; 66m; 92m

Found in 0.210225ms

Beispiel 10

Eingabe:

- Beispiel 2

Ausgabe:

6 people attend for the lengths: 10m; 60m; 90m

Found in 0.17484ms

Beispiel 11

Eingabe:

- Beispiel 1

Ausgabe:

6 people attend for the lengths: 22m; 51m; 64m

Found in 0.175155ms

Quellcode

```
MyBitSet[] bitSets = initializeBitSets(values, persons);

final MyBitSet preComputed = new MyBitSet(bitsetLength);

int len = values.length;
for (int i = 0; i < len - 2; i++) {
    for (int j = len - 2; j >= i + 1; j--) {
        preComputed.copyOr(bitSets[i], bitSets[j]);
        for (int k = j + 1; k < len; k++) {
            int currCount = preComputed.orCardinality(bitSets[k]);
            if (currCount > bestCount) {
                bestCount = currCount;
                bestLengths[0] = values[i];
                bestLengths[1] = values[j];
                bestLengths[2] = values[k];
            } else if (currCount + (len - k - 1) + doubledCount <= bestCount) {
```

```
                break;
            }
        }
    }
}

public class MyBitSet {
    private final long[] words;

    public void copyOr(MyBitSet first, MyBitSet second) {
        final long[] firstWords = first.words;
        final long[] secondWords = second.words;
        for (int i = 0; i < words.length; i++) {
            this.words[i] = firstWords[i] | secondWords[i];
        }
    }

    public void set(final int bitIndex) {
        words[bitIndex >> 6] |= (1L << bitIndex);
    }

    public int orCardinality(final MyBitSet other) {
        int count = 0;
        final long[] otherWords = other.words;
        final int length = words.length;
        for (int i = 0; i < length; i++) {
            count += Long.bitCount(words[i] | otherWords[i]);
        }
        return count;
    }
}
```