Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Institut für Theoretische Informatik
Peter Widmayer
Thomas Tschager
Antonis Thomas

13. April 2016

Datenstrukturen & Algorithmen Lösungen zu Blatt 6 FS 16

Lösung 6.1 *Marsmission.*

Definition der DP-Tabelle: Wir benutzen eine $(m \times n)$-Tabelle T. Der Eintrag $T[i][j]$ enthält den maximal erreichbaren Wert der Gesteinsproben auf einem Süd-Ost-Weg von $(1,1)$ zu (i,j).

Berechnung eines Eintrags: Für (i,j) mit $1 < i,j \leq n$ beobachten wir folgendes: Endet ein Weg in der Position (i,j), dann sind wir entweder von oben oder von links gekommen. Also ist der maximal erreichbare Wert genau

$$T[i,j] = A[i,j] + \max\{T[i-1,j], T[i,j-1]\}, \tag{1}$$

denn zum Wert $A[i,j]$ der Gesteinsprobe auf der Position (i,j) muss noch der maximale Wert der Gesteinsproben auf den vorherigen Positionen addiert werden. Für die Fälle am oberen und am linken Rand legen wir folgendes fest:

- $T[1,1] = A[1,1] = 0$, denn $(1,1)$ ist die Startposition,
- $T[i,1] = A[i,1] + T[i-1,1]$ für $i > 1$, denn die Position $(i,1)$ kann nur von der Position oben, $(i-1,1)$, erreicht werden,
- $T[1,j] = A[1,j] + T[1,j-1]$ für $j > 1$, denn die Position $(1,j)$ kann nur von der Position links, $(1,j-1)$, erreicht werden.

Berechnungsreihenfolge: Der Eintrag $T[i,j]$ hängt nur von Einträgen für kleinere Werte von i und j ab. Daher können die Einträge zum Beispiel nach aufsteigenden Werten von $i = 1,\ldots,m$ und für gleiche i jeweils nach aufsteigenden Werten von $j = 1,\ldots,n$ berechnet werden.

Auslesen der Lösung: Der maximal erreichbare *Wert* der Gesteinsproben ist am Ende im Eintrag $T[m,n]$ gespeichert. Zur Rekonstruktion des Weges starten wir im Eintrag $T[m,n]$ und geben (m,n) aus. Danach wird geprüft, ob $T[m,n] = A[m,n] + T[m-1,n]$ ist. Dann sind wir von oben, also vom Eintrag $(m-1,n)$, gekommen und fahren entsprechend dort fort. Ansonsten ist $T[m,n] = A[m,n] + T[m,n-1]$ und wir sind von links gekommen. Also fahren wir beim Eintrag $(m,n-1)$ fort. Die Rekonstruktion wird so fortgesetzt, bis die Startposition $(1,1)$ erreicht wird. Wird vorzeitig der Rand der Tabelle (links oder oben) erreicht, dann wird die Rekonstruktion beim einzig möglichen Vorgängerfeld (oben oder links) fortgesetzt. Am Ende haben wir die Positionen des Weges in umgekehrter Reihenfolge ermittelt.

Laufzeit: Die Tabelle besitzt Grösse $m \cdot n$, und jeder Eintrag kann in Zeit $\mathcal{O}(1)$ berechnet werden. Damit kann die Berechnung des maximalen Werts in Zeit $\mathcal{O}(mn)$ erfolgen.

Der rekonstruierte Weg besitzt Länge $m + n - 1$. Da für jede Position in Zeit $\mathcal{O}(1)$ entschieden werden kann, ob wir von oben oder von links gekommen sind, kann der gesamte Weg in Zeit $\mathcal{O}(m+n)$ ermittelt werden. Zusammen mit der Zeit zum Ausfüllen der Tabelle ergibt sich also eine Gesamtlaufzeit von $\mathcal{O}(mn)$.

Lösung 6.2 *Selbstanordnende Listen.*

Zugriff auf 'R': 5 Vergleiche. Neue Struktur der Liste:

$$\mathrm{R} \longrightarrow \mathrm{A} \longrightarrow \mathrm{L} \longrightarrow \mathrm{G} \longrightarrow \mathrm{O} \longrightarrow \mathrm{I} \longrightarrow \mathrm{T} \longrightarrow \mathrm{H} \longrightarrow \mathrm{M} \longrightarrow \mathrm{U} \longrightarrow \mathrm{S}$$

Zugriff auf 'A': 2 Vergleiche. Neue Struktur der Liste:

$$\mathrm{A} \longrightarrow \mathrm{R} \longrightarrow \mathrm{L} \longrightarrow \mathrm{G} \longrightarrow \mathrm{O} \longrightarrow \mathrm{I} \longrightarrow \mathrm{T} \longrightarrow \mathrm{H} \longrightarrow \mathrm{M} \longrightarrow \mathrm{U} \longrightarrow \mathrm{S}$$

Zugriff auf 'I': 6 Vergleiche. Neue Struktur der Liste:

$$\mathrm{I} \longrightarrow \mathrm{A} \longrightarrow \mathrm{R} \longrightarrow \mathrm{L} \longrightarrow \mathrm{G} \longrightarrow \mathrm{O} \longrightarrow \mathrm{T} \longrightarrow \mathrm{H} \longrightarrow \mathrm{M} \longrightarrow \mathrm{U} \longrightarrow \mathrm{S}$$

Zugriff auf 'S': 11 Vergleiche. Neue Struktur der Liste:

$$\mathrm{S} \longrightarrow \mathrm{I} \longrightarrow \mathrm{A} \longrightarrow \mathrm{R} \longrightarrow \mathrm{L} \longrightarrow \mathrm{G} \longrightarrow \mathrm{O} \longrightarrow \mathrm{T} \longrightarrow \mathrm{H} \longrightarrow \mathrm{M} \longrightarrow \mathrm{U}$$

Zugriff auf 'T': 8 Vergleiche. Neue Struktur der Liste:

$$\mathrm{T} \longrightarrow \mathrm{S} \longrightarrow \mathrm{I} \longrightarrow \mathrm{A} \longrightarrow \mathrm{R} \longrightarrow \mathrm{L} \longrightarrow \mathrm{G} \longrightarrow \mathrm{O} \longrightarrow \mathrm{H} \longrightarrow \mathrm{M} \longrightarrow \mathrm{U}$$

Zugriff auf 'A': 4 Vergleiche. Neue Struktur der Liste:

$$\mathrm{A} \longrightarrow \mathrm{T} \longrightarrow \mathrm{S} \longrightarrow \mathrm{I} \longrightarrow \mathrm{R} \longrightarrow \mathrm{L} \longrightarrow \mathrm{G} \longrightarrow \mathrm{O} \longrightarrow \mathrm{H} \longrightarrow \mathrm{M} \longrightarrow \mathrm{U}$$

Zugriff auf 'A': 1 Vergleich. Neue Struktur der Liste:

$$\mathrm{A} \longrightarrow \mathrm{T} \longrightarrow \mathrm{S} \longrightarrow \mathrm{I} \longrightarrow \mathrm{R} \longrightarrow \mathrm{L} \longrightarrow \mathrm{G} \longrightarrow \mathrm{O} \longrightarrow \mathrm{H} \longrightarrow \mathrm{M} \longrightarrow \mathrm{U}$$

Zugriff auf 'G': 7 Vergleiche.

Zusammen ergeben sich also $5 + 2 + 6 + 11 + 8 + 4 + 1 + 7 = 44$ Vergleiche.

Lösung 6.3 *Tauschgeschäfte (**Programmieraufgabe**).*

Gegeben sind n verschiedene Arten von Gegenständen, wobei a_i die Anzahl der Gegenstände der Art i ist für $1 \leq i \leq n$. Wir können das Problem lösen, indem wir jeweils drei Gegenstände der drei am häufigsten vorkommenden Arten gegen einen Franken eintauschen, solange es noch drei verschiedene Gegenstände gibt. Wir bezeichnen die Gesamtanzahl aller Gegenstände mit $sum = \Sigma_{i=1}^{i=n} a_i$ und die Häufigkeit einer am häufigsten vorkommenden Art von Gegenständen mit $max = \max_i a_i$.

In einem ersten Ansatz verwenden wir ein Array `freq` mit `freq[i]` $= a_i$ für alle i. Wir sortieren das Array und tauschen drei Gegenstände der drei häufigsten Arten, indem wir die drei letzten Einträge des Arrays je um 1 verringern. Anschliessend muss das Array neu sortiert werden. Diesen Tauschvorgang wiederholen wir solange wie möglich und geben schliesslich die Anzahl der erhaltenen Franken aus. Wir erhalten höchstens $\lfloor \frac{sum}{3} \rfloor$ Franken, da jeweils drei verschiedene Gegenstände getauscht werden müssen. Es gilt aber auch $sum \leq n \cdot max$. Daher sind $\mathcal{O}(n \cdot max)$ Tauschvorgänge möglich. Nach jedem Tauschvorgang muss das Array `freq` neu sortiert werden (in $\mathcal{O}(n \log n)$ Laufzeit). Daher ist die Gesamtlaufzeit $\mathrm{O}(n^2 \log(n) \cdot max)$.

```
import java.util.Scanner;
import java.util.Arrays;

class Main {

    static int N = 5000 + 1;

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int test = in.nextInt();

        for (int t = 0; t < test; t++) {
            int n = in.nextInt();
            //array of the frequencies read at input
            int[] freq = new int[n];
            for (int i = 0; i < n; i++) {
                int a = in.nextInt();
                freq[i] = a;
            }

            int answer = 0;

            Arrays.sort(freq);
            while (n > 2 && freq[n-3] > 0) {
                answer++;
                freq[n-1]--; freq[n-2]--;freq[n-3]--;
                Arrays.sort(freq);
            }

            System.out.println(answer);
        }
    }
}
```

Eine effizientere Lösung erhalten wir, wenn wir ein Array `Bucket` verwenden, wobei `Bucket[k]` angibt, von wie vielen verschiedenen Arten es genau `k` Gegenstände gibt. Wir merken uns die Häufigkeit der drei am häufigsten vorkommenden Arten von Gegenständen max_1, max_2 und max_3. Bei einem Tauschvorgang dekrementieren wir $\texttt{Bucket}[\texttt{max}_\texttt{j}]$ um 1 und inkrementieren $\texttt{Bucket}[\texttt{max}_\texttt{j} - \texttt{1}]$ um 1 für alle $1 \leq j \leq 3$. Danach aktualisieren wir max_1, max_2 und max_3: Solange für ein $1 \leq j \leq 3$ gilt, dass $\texttt{Bucket}[\texttt{max}_\texttt{j}] == 0$, wird max_j um 1 dekrementiert. Wir müssen allerdings beachten, dass genügend verschiedenartige Gegenstände vorhanden sind, wenn $max_j == max_k$ für $1 \leq k < j \leq 3$. Daher prüfen wir für beliebige $j \in \{2, 3\}$, dass $\texttt{Bucket}[\texttt{max}_\texttt{j}] \geq 2$ sein muss, falls $max_j == max_{j-1}$ gilt. Andernfalls muss max_j weiter dekrementiert werden. Weiters muss $\texttt{Bucket}[\texttt{max}_\texttt{3}] \geq 3$ sein, falls $max_1 == max_2 == max_3$ gilt.

Durch die Verwendung des Arrays `Bucket` vermeiden wir die Sortierung nach jedem Tauschvorgang. Die Laufzeit ist daher in $\mathcal{O}(n \cdot max)$. Eine letzte Beobachtung kann unsere Laufzeit weiter beschleunigen:

Behauptung: Wenn $sum \geq 3max$, dann können $\lfloor \frac{sum}{3} \rfloor$ Franken getauscht werden.

Beweis (induktiv): Bei einem Tauschvorgang wählen wir die drei häufigsten Arten und dekrementieren deren Anzahl jeweils um 1. Ändert sich nach dem Tauschvorgang die Häufigkeit der häufigsten Art, dann wissen wir dass max um 1 verringert wurde und sum um 3, d.h. die Invariante bleibt bestehen. Bleibt max allerdings unverändert, dann gab es vor dem Tauschvorgang mindestens 4 Arten mit je max Gegenständen. Nach dem Tauschvorgang gibt es mindestens 3 Arten mit $max - 1$ Gegenständen und eine Art mit max Gegenständen. Die Gesamtanzahl dieser 4 Arten ist also mindestens $4 \cdot max - 3$ und somit immer noch grösser als $3max$, falls $max \neq 1$ und $max \neq 2$ gilt. Wenn $max = 1$ gilt, gab es vor dem letzten Tauschvorgang von jeder Art nur höchstens ein Element, d.h. $4 \leq sum$ und $3 \max = 3$. Bei $max = 2$ gab es 4 Arten mit je zwei Gegenständen und es können $\lfloor \frac{sum}{3} \rfloor = \lfloor \frac{8}{3} \rfloor$ Franken getauscht werden.

Wenn wir diese Behauptung berücksichtigen, können wir zuerst überprüfen ob $sum \geq 3max$. In diesem Fall können wir die Lösung sofort ausgeben. Andernfalls führen wir den oben beschriebenen Algorithmus aus, wobei die Laufzeit in $\mathcal{O}(max)$ ist, weil in diesem Fall nur $\mathcal{O}(max)$ Tauschvorgänge möglich sind. Damit ist die Laufzeit in jedem Fall in $O(max)$.

Die Behauptung ist auch hilfreich um zu zeigen, dass dieser Algorithmus die korrekte Lösung liefert. Wir unterscheiden drei Fälle:

1. $sum \geq 3 \cdot max$: Wir wissen wegen der Behauptung bereits, dass unser Algorithmus korrekt ist.

Andernfalls gilt $sum < 3 \cdot max$ und die Lösung ist kleiner als max. Deshalb können wir in jedem Tauschvorgang einen Gegenstand der häufigsten Art wählen. Wir müssen also lediglich zwei verbleibenden Gegenstände für jeden Tauschvorgang wählen. Das führt zum vereinfachten Teilproblem, bei dem je zwei verschiedenartige Gegenstände getauscht werden müssen. Wir berücksichtigen die häufigste Art von Gegenständen der Eingabe nicht, da diese ja in jedem Tauschvorgang gewählt wird. Wir bezeichnen mit sum_r die Gesamtanzahl der Gegenstände für dieses Teilproblem, d.h. die Anzahl der Gegenstände ohne die Gegenstände der häufigsten Art. Mit max_r bezeichnen wir die Häufigkeit der am häufigsten vorkommenden Art für dieses Teilproblem. Wir können nun unsere Behauptung von oben für das Teilproblem anpassen: *Wenn* $sum_r \geq 2 \cdot max_r$ *dann können* $sum_r/2$ *Franken getauscht werden.* Die Behauptung kann auf dieselbe Weise bewiesen werden wie die ursprüngliche Behauptung oben. Das führt zu den letzten beiden Fällen:

2. $sum < 3 \cdot max$ und $sum_r \geq 2 \cdot max_r$: Wegen der obigen Behauptung für das vereinfachte Teilproblem liefert der Ansatz ebenfalls die optimale Lösung.
3. $sum < 3 \cdot max$ und $sum_r < 2 \cdot max_r$: Hier gilt wiederum, dass für das Teilproblem höchstens max_r Tauschvorgänge durchgeführt werden können, wobei in jedem Tauschvorgang einer der max_r Gegenstände der häufigsten Art im Teilproblem gewählt wird.

Daher liefert unser Greedy-Ansatz eine korrekte Lösung.

```
import java.util.Scanner;

class Main{

    static int N = 5000+1;

    public static void main(String [] args){
        Scanner in = new Scanner(System.in);
        int test = in.nextInt();

        for (int t = 0; t < test; t++){
            int n = in.nextInt();
            //For each testcase we have an array called Bucket.
            //Bucket[k] represents how many different item types
            //have k items.
            int [] Bucket = new int [N];
            int max = 0, sum = 0;
            for(int i=0;i<n;i++){
                int a = in.nextInt();
                if (max<a) max =a;
                sum+=a;
                Bucket[a]++;
            }

            //This is the condition from the Claim.
            if (sum>=3*max) { System.out.println(sum/3); continue; }

            //In this loop we find the three types that have the largest
            //number of items.
            int max1 = -1, max2 = -1, max3 = -1;
            for(int i=max;i>=0;i--) {
                if (Bucket[i] > 0) {
                    if(max1 == -1) max1 = i;
                    else if(max2 == -1) max2 = i;
                    else max3 = i;

                    Bucket[i]--;
                    if(max3 != -1) break;
                    //By increasing i we make sure that i will be examined again
                    // in the case there was more than one types that
                    // have i as the number of items.
                    i++;
                }
            }
            //Since we reduced the Bucket value of each of the three maxes above
            // we increase it back to its right value.
            if (max1 > 0) Bucket[max1]++; if (max2 > 0) Bucket[max2]++; if (max3 >
                 0) Bucket[max3]++;

            //We are ready to calculate the answer now.
            int answer = 0;
            //While max3 > 0 it means that we have at least 3 different types
                available.
            while(max3 > 0) {
                answer++;
                //Update the values of Bucket;
                Bucket[max1]--;Bucket[max2]--;Bucket[max3]--;
                Bucket[max1-1]++;Bucket[max2-1]++;Bucket[max3-1]++;

                //Update the new maxes.
                while(max1 > 0 && Bucket[max1]==0) max1--;
                while(max2 > 0 && Bucket[max2]==0) max2--;
                //In this case there is only one type in Bucket[max2] and since
                    max1 == max2 we have to
                //reduce max2 further. Similarly below for max3.
                if(Bucket[max2] == 1 && max1 == max2) {max2--; while(max2 > 0 &&
                    Bucket[max2]==0) max2--;}
                while(max3 > 0 && Bucket[max3]==0) max3--;
                if(Bucket[max3] == 1 && max3 == max1) { max3--; while(max3 > 0 &&
                    Bucket[max3]==0) max3--;}
                if(Bucket[max3] == 1 && max3 == max2) { max3--; while(max3 > 0 &&
                    Bucket[max3]==0) max3--;}
                if(Bucket[max3] == 2 && max3 == max2 && max3 == max1) { max3--;
                    while(max3 > 0 && Bucket[max3]==0) max3--;}
            }
            System.out.println(answer);
        }
    }
}
```