37. Bundeswettbewerb Informatik, 1. Runde

Erik Sünderhauf

Aufgabe 3: Voll daneben

Teilnehmer-ID: 51468; Team-ID: 00921

1 Lösungsidee

Diese Aufgabe lässt sich mittels dynamischer Programmierung lösen. Im folgenden sei n die Anzahl der Teilnehmer an dem Gewinnspiel und k die Anzahl der Zahlen die der Casinobesitzer wählt. Weiterhin ist $dp_{j,i}$ die minimal auszuzahlende Summe, wenn Al Capone i Zahlen optimal auf die ersten j Glückszahlen verteilt.

Die Glückszahlen bilden dabei eine Folge $a_1, a_2, \cdots, a_n$, mit $a_1 \leq a_2 \leq \cdots \leq a_n$. Die Von Al Capone gewählten Zahlen bilden die Folge $b_1, b_2, \cdots, b_k$ mit $b_1 \leq b_2 \leq \cdots \leq b_k$. Das beide Folgen aufsteigend sortiert sind lässt sich dabei ohne Beschränkung der Allgemeinheit annehmen.

Ein Paar (i, j) ganzer Zahlen soll genau dann *verbunden* genannt werden, wenn gilt:

$$\nexists l \in \mathbb{N}, 1 \leq l \leq k : \begin{cases} |a_i - b_l| < |a_i - b_j| \\ |a_i - b_l| = |a_i - b_j|, l < j \end{cases}$$

Außerdem soll die Indexmenge I genau dann eine *Gruppe* genannt werden, wenn gilt:

$$\exists j \in \mathbb{N}, 1 \leq j \leq k,\ \forall i \in I : (i, j) \text{ sind verbunden.}$$

Weiterhin gehört die *Gruppe* I zu j, wenn j die Bedingungen der Definition der Grupper erfüllt.

Offensichtlich führt die Wahl einer beliebigen Folge b_i dazu, dass die Glückszahlen in Gruppen unterteilt werden. Nach Definition sind zwei Gruppen disjunkt, weshalb sich die auszuzahlende Summe wie folgt ermitteln lässt. Dabei ist $G = \{I | I \text{ ist eine Gruppe}\}$ die Menge aller Gruppen.

$$\sum_{g} \sum_{i \in I_g} |a_i - b_j|,\ I_g \in G \text{ und } I_g \text{ gehört zu } j$$

Die innere Summe ist dabei nur von der jeweiligen Gruppe abhängig. Aus diesem Grund genügt es diese lokale Summe zu minimieren und eine optimale Unterteilung in Gruppen zu finden um eine möglichst geringe auszuzahlende Summe zu erhalten. Im folgenden soll auf diese beiden Aspekte eingegangen werden.

Um die lokale Summe $\sum_{i \in I_g} |a_i - b_j|$ zu minimieren genügt folgendes Lemma.

Lemma 1.1. *Sei S eine beliebige endliche Menge reeller Zahlen. Der Ausdruck*

$$\sum_{s \in S} |s - x|$$

wird minimal, wenn x dem Median der Menge S entspricht.

Beweis. Sei $n = |S|$ die Anzahl der Elemente in der Menge und $s_i \in S$ für $1 \leq i \leq n$, sodass $s_1 \leq s_2 \leq \cdots \leq s_n$. Es werden nun zwei Fälle unterschieden.

Fall 1 $n \equiv 1 \mod 2$

Die Summe lässt sich wie folgt umschreiben:

$$\sum_{i=1}^{n} |s_i - x| = \sum_{i=2}^{n-1} |s_i - x| + |s_n - x| + |x - s_1| \tag{1}$$

Es ist offensichtlich optimal, wenn $s_1 \leq x \leq s_n$, da sich für $x > s_n$ oder $x < s_1$ der Ausdruck in 1 wie folgt nach unten abschätzen lässt.

$$\sum_{i=2}^{n-1} |s_i - x| + |s_n - x| + |x - s_1| > \sum_{i=2}^{n-1} |s_i - x| + s_n - s_1 \tag{2}$$

Für $s_1 \leq x \leq s_n$ ist 1 aber äquivalent zu:

$$\sum_{i=2}^{n-1} |s_i - x| + |s_n - x| + |x - s_1| = \sum_{i=2}^{n-1} |s_i - x| + s_n - s_1 \tag{3}$$

Dieser Prozess lässt sich fortführen und man erhält:

$$\sum_{i=1}^{n} |s_i - x| = |s_{\frac{n+1}{2}} - x| + \sum_{i=1}^{\frac{n-1}{2}} s_i - s_{n-i+1} \tag{4}$$

Wie man leicht sieht, wird 4 für $x = s_{\frac{n+1}{2}}$ minimal. Also entspricht in diesem Fall x dem Median der Menge.

Fall 2 $n \equiv 0 \mod 2$

Mit obiger Begründung lässt sich die gegebene Summe wie folgt umschreiben:

$$\sum_{i=1}^{n} |s_i - x| = |s_{\frac{n}{2}} - x| + |s_{\frac{n+2}{2}} - x| + \sum_{i=1}^{\frac{n-2}{2}} s_i - s_{n-i+1} \tag{5}$$

Wie im ersten Fall bereits gezeigt wurde, nimmt 5 ein Minimum für $s_{\frac{n}{2}} \leq x \leq s_{\frac{n+2}{2}}$ an. O.B.d.A. wählt man $x = s_{\frac{n}{2}}$, also dem Median der Menge.

Durch Betrachtung aller Fälle wurde gezeigt, dass die Summe minimal wird, wenn x dem Median der Menge S entspricht. $\square$

Nun genügt es eine optimale Partitionierung der Indexmenge in *Gruppen* zu finden, sodass die auszuzahlende Summe minimal ist. Nach Definition gilt für zwei beliebige *Gruppen* I_1 und I_2 einer Partitionierung:

$$\max I_1 < \min I_2 \text{ bzw. } \max I_2 < \min I_1$$

Dazu ist es sinnvoll die anfangs erwähnte dp Funktion zu verwenden, da eine optimale Unterteilung der ersten n Zahlen in disjunkte Teilintervalle gefragt ist. Dabei ist die Funktion wie folgt rekursiv definiert:

$$dp_{j,i} = \min_{1 \leq l < i} (dp_{j-1,l} + C_{l+1,i})$$

Dabei ist $C_{l+1,i}$ das Minimum der lokalen Summe, welcher ihr Minimum nach Lemma 1.1 annimmt.

Es genügt also $dp_{k,n}$ zu berechnen und sich zu jedem Zustand den optimalen Index l zu speichern, wodurch man die minimal auszuzahlende Summe und eine Parition in Gruppen erhält. Nach Lemma 1.1 lassen sich aus dieser Partition die Zahlen bestimmen, welche Al Capone wählen sollte.[1]

1.1 Theoretische Analyse

Um die *dp* Funktion zu berechnen, müssen $\mathcal{O}(nk)$ Zustände berechnet werden. Um einen Zustand zu berechnen, werden $\mathcal{O}\left(n^2\right)$ Rechenoperationen bei einer naiven Implementierung benötigt. Dies setzt sich aus dem Bestimmen der Kostenfunktion $C_{l+1,i}$ ($\mathcal{O}(n)$) und dem Finden des Index l zusammen ($\mathcal{O}(n)$).

Da die Folge a_i der Glückszahlen statisch ist, lässt sich das Minimum der lokalen Summe nach Lemma 1.1 mittels Präfix Summen in $\mathcal{O}(1)$ bestimmen. Damit wurde die Laufzeit auf $\mathcal{O}\left(n^2k\right)$ begrenzt.

Sei $opt_{j,i}$ eine Funktion, welche ähnlich wie die *dp* Funktion definiert ist.

$$opt_{j,i} = \arg\min_{1 \leq l < i} (dp_{j-1,l} + C_{l+1,i})$$

Diese Funktion gibt im Grunde genommen den Index des Medians der j-ten Gruppe einer optimalen Partition an. Fügt man nun der j-ten Gruppe das Element a_{i+1} hinzu, so folgt, da $a_i \leq a_{i+1}$:

$$opt_{j,i} \leq opt_{j,i+1}$$

Diese Ungleichung ist offensichtlich gültig, da durch das Hinzufügen eines Elementes der Index des Medians niemals kleiner werden kann. Dies wäre nur dann der Fall, wenn zu der j-ten Gruppe ein Element kleiner als der Median hinzugefügt wird, was aber ein Widerspruch zur Optimalität der j-ten Gruppe vor dem Hinzufügen ist. Aus der Gültigkeit der Ungleichung folgt, dass sich die Divide and Conquer Optimization anwenden lässt, wodurch die Laufzeit nur noch $\mathcal{O}(nk\log n)$ beträgt.

[1] Siehe Code für mehr Details.

2 Umsetzung

Die Lösungsidee wird in C++11 implementiert. Dazu werden erst die Daten in dem gegebenen Format aus der Datei *input.txt* gelesen. Anschließend wird die Zahlenfolge sortiert und die DP Funktion berechnet. Zum Schluss werden die minimal auszuzahlende Summe und die Zahlen, welche Al Capone wählen sollte, ausgegeben. Dabei wird erwartet, dass in der ersten Zeile die beiden ganzen Zahlen n und k stehen und in der nächsten Zeile die n Glückszahlen.

3 Beispiele

3.1 Beispiel 1

Datei: "input1.txt"
Ausgabe: 4950
945 840 735 630 525 430 335 240 145 50

3.2 Beispiel 2

Datei: "input2.txt"
Ausgabe: 1924
929 862 777 651 539 421 368 315 172 59

3.3 Beispiel 3

Datei: "input3.txt"
Ausgabe: 2160
960 860 720 660 580 520 440 340 240 100

3.4 Beispiel 4

Datei: "input4.txt"
Ausgabe: 1
999 8 6 4 2

3.5 Beispiel 5

Datei: "input5.txt"
Ausgabe: 44
105 87 70

3.6 Beispiel 6

Datei: "input6.txt" $(n = 4 \cdot 10^4, k = 2 \cdot 10^2)$
Ausgabe: 2000000
39900 39699 39498 39297 39096 38895 38694 38493 38292 38091 37890 37689

37488 37287 37086 36885 36684 36483 36282 36081 35880 35679 35478 35277 35076 34875 34674 34473 34272 34071 33870 33669 33468 33267 33066 32865 32664 32463 32262 32061 31860 31659 31458 31257 31056 30855 30654 30453 30252 30051 29850 29649 29448 29247 29046 28845 28644 28443 28242 28041 27840 27639 27438 27237 27036 26835 26634 26433 26232 26031 25830 25629 25428 25227 25026 24825 24624 24423 24222 24021 23820 23619 23418 23217 23016 22815 22614 22413 22212 22011 21810 21609 21408 21207 21006 20805 20604 20403 20202 20001 19801 19602 19403 19204 19005 18806 18607 18408 18209 18010 17811 17612 17413 17214 17015 16816 16617 16418 16219 16020 15821 15622 15423 15224 15025 14826 14627 14428 14229 14030 13831 13632 13433 13234 13035 12836 12637 12438 12239 12040 11841 11642 11443 11244 11045 10846 10647 10448 10249 10050 9851 9652 9453 9254 9055 8856 8657 8458 8259 8060 7861 7662 7463 7264 7065 6866 6667 6468 6269 6070 5871 5672 5473 5274 5075 4876 4677 4478 4279 4080 3881 3682 3483 3284 3085 2886 2687 2488 2289 2090 1891 1692 1493 1294 1095 896 697 498 299 100
Laufzeit: 429ms

4 Quellcode

Einlesen und Verarbeiten der Daten:

```
int n, k;
scanf("%d %d", &n, &k);
for (int i = 1; i <= n; i++)
  scanf("%d", &a[i]);
sort(a + 1, a + n + 1);
for (int i = 0; i <= n; i++) {
  dp[0][i] = INF, dp[1][i] = INF;
  if (i)
    sum[i] = sum[i - 1] + a[i]; // präfix summe
}
```

Kostenfunktion $C_{l+1,i}$:

```
int f(int l, int r, int m) {
// Summe der Elemente-Median von l bis r
  if (l > r)
    return 0;
  return sum[r] - sum[l - 1] - (r - l + 1) * a[m];
}

int cost(int i, int j) {
// gibt das Minimum der lokalen Summe nach Lemma 1.1 zurück
// return (summe der Elemente größer als der Median - Median) -
// (Summe der Elemente kleiner als der Median - Median)
  return f((i + j) / 2 + 1, j, (i + j) / 2) -
    f(i, (i + j) / 2 - 1, (i + j) / 2);
}
```

Berechnen des DP-Arrays mit Divide and Conquer Optimization:

```
void dfs(int j, int a, int b, int oA, int oB) {
  if (a > b)
    return;
  int m = (a + b) / 2;
  // divide and conquer optimization
  for (int l = oA; l <= min(oB, m - 1); l++) {
    // berechne dp[j][m]
    int v = dp[(j & 1) ^ 1][l] + cost(l + 1, m);
    if (v < dp[j & 1][m]) {
      dp[j & 1][m] = v;
      par[j][m] = l;
    }
  }
  dfs(j, a, m - 1, oA, par[j][m]);
  dfs(j, m + 1, b, par[j][m], oB);
}

// Aufruf der Funktion dfs
dp[0][0] = 0;
for (int j = 1; j <= k; j++)
  // berechne das dp Array layer by layer
  dfs(j, 1, n, 0, n - 1);
```