Aufgabe 3: Eisbudendilemma

Teilnahme-Id: 57429

Bearbeiter dieser Aufgabe:
Lucas Schwebler

18. April 2021

Inhaltsverzeichnis

1 Lösungsidee **2**
1.1 Situation zwischen zwei Eisbuden . . . 2
1.1.1 Keine Eisbude . . . 2
1.1.2 Zwei oder mehr Eisbuden . . . 2
1.1.3 Eine Eisbude . . . 3
1.2 Vorberechnungen . . . 5
1.2.1 Präfixsummen . . . 5
1.2.2 Maximale Intervalle . . . 5
1.2.3 Nutzen der Vorberechnungen . . . 5
1.3 Beispiel zur Veranschaulichung . . . 6
1.4 Optimierungen . . . 6
1.4.1 Vorberechnung mit Monotonie-Trick . . . 6
1.4.2 Eine der Eisbuden steht direkt bei einem Haus . . . 8
1.4.3 Frühzeitiges Abbrechen der Suche . . . 9

2 Unabhängigkeit vom Umfang **9**
2.1 Überblick . . . 10
2.2 Maximale Intervallgröße . . . 10
2.3 Finden der genauen Position . . . 10
2.4 Veranschaulichung . . . 11
2.5 Lösung in Linearzeit . . . 11

3 Erweiterung **13**

4 Umsetzung **14**
4.1 Arithmetik . . . 14
4.2 Vorberechnung . . . 14
4.3 Stabilität testen . . . 15
4.4 Testen der Kombinationen . . . 15

5 Beispiele **16**
5.1 BWINF-Beispiele . . . 16
5.2 Eigene Beispiele . . . 17
5.3 Empirische Betrachtung . . . 18

6 Quellcode **19**
6.1 Allgemein . . . 19
6.2 Einfache Implementation . . . 20
6.3 Optimierte Implementation . . . 20
6.4 Lineare Lösung . . . 22
6.5 Erweiterung . . . 24

Vorbemerkung Diese Einsendung enthält 3 Lösungen zur Aufgabe. Dabei kann die zweite als Optimierung der ersten gesehen werden. Die dritte verwendet ein leicht anderes Konzept, baut aber dennoch stark auf den Beobachtungen der ersten beiden Lösungen auf.

1 Lösungsidee

Gegeben ist der Umfang u des Sees sowie n Adressen von Häusern. Das Ziel ist es, eine stabile Verteilung der Eisbuden zu finden. Der wohl offensichtlichste Ansatz ist das Ausprobieren aller möglichen Eisbudenverteilungen, wobei für jede getestet wird, ob sie stabil ist. Insgesamt gibt es $\mathcal{O}(u^3)$ mögliche Eisbudenverteilungen, da drei Eisbuden gewählt werden, die sich jeweils bei einer ganzzahligen Position von 0 bis $u-1$ befinden können. Würde man nun pro Eisbudenverteilung naiv alle $\mathcal{O}(u^3)$ möglichen Vorschläge durchprobieren, wäre die Laufzeit mindestens $\mathcal{O}(u^6)$. Dies wäre für praktische Anwendung auf die Beispiele der BWINF-Webseiten viel zu langsam. Stattdessen soll ein Ansatz entwickelt werden, mit dem effizient bestimmt werden kann, ob eine Kombination der Eisbuden stabil ist. Es wird sich herausstellen, dass dies mit ein paar Vorberechnungen in $\mathcal{O}(1)$ möglich ist. Um diesen Ansatz zu entwickeln, soll folgendes Optimierungsproblem gelöst werden:

Finde den Vorschlag, der bei einer Abstimmung die meisten Stimmen erhalten würde.

Der Zustand ist genau dann stabil, wenn die Lösung des Optimierungsproblems nicht der Mehrheit entspricht, also $\leq \frac{n}{2}$ ist.

Offensichtlich ist es nicht zielführend, wenn ein Vorschlag die Position einer Eisbude unverändert lässt. Denn so wird für keinen Dorfbewohner die Distanz zu dieser Eisbude geringer – sie bleibt gleich. Somit ist es nur sinnvoll Vorschläge zu betrachten, bei denen die neuen Eisbuden zwischen zwei aktuellen Eisbuden platziert werden. Nun soll betrachtet werden, wie viele Stimmen sich mit wie vielen Eisbuden in diesen Zwischenräumen holen lassen.

1.1 Situation zwischen zwei Eisbuden

Im folgenden soll das Intervall untersucht werden, das von 2 Eisbuden abgegrenzt wird, wobei keine weitere Eisbude zwischen diesen existiert. Es wird o.B.d.A. angenommen, dass Eisbude 1 bei 0 und Eisbude 2 bei $m > 0$ steht. Die Positionen können nämlich in diesen Bereich transformiert werden: Von den beiden Positionen wird die erste abgezogen. Danach befinden sich beide bei 0 bzw. m befinden. Somit lassen sich alle Kombinationen von zwei Eisbudenpositionen auf diesen, hier zum Rechnen vereinfachten, Fall reduzieren.

1.1.1 Keine Eisbude

Wenn keine neue Eisbude in diesem Intervall platziert wird, werden alle Bewohner des Intervalls gegen den Vorschlag stimmen. Die alten Eisbudenpositionen bei 0 und m werden dann zwischen dem Haus und den neuen Eisbuden liegen. Somit ist der Abstand zur nächsten Eisbude größer geworden, weshalb dagegen gestimmt wird.

1.1.2 Zwei oder mehr Eisbuden

Sollen bei einem Vorschlag zwei neue Eisbuden in $I = [1; m-1]$ platziert werden, so ist es möglich, sie so zu wählen, dass alle Bewohner von I diesen Vorschlag annehmen werden. Dazu können die neuen Eisbuden bei 1 und $m-1$ platziert werden. Denn zwischen den bisherigen Eisbuden gibt es keine weitere, womit für alle Bewohner von I die nächste Eisbude entweder bei 0 oder m war. Somit ist die neue nächste Eisbude für diese nun um eins näher (bei 1 oder $m-1$). Die Anzahl der Stimmen, die so geholt werden kann, wird ab jetzt durch den Ausdruck *sum* beschrieben.

Zur Veranschaulichung wird eine Situation aus Beispiel 1 verwendet. Bei H steht ein Haus und bei E eine Eisbude. In grau (E) sind die alten Eisbuden und die neuen sind dick gedruckt (**E**). Offensichtlich verringert sich der Abstand zur nächsten Eisbude für alle Häuser von 1 bis 12:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
H		H	H					H				H		H	H				
E	**E**											**E**	E						

Werden 2 Eisbuden auf ein solches Intervall aufgeteilt, holt der optimale Vorschlag also alle Stimmen in diesem Intervall. Es ist somit auch nicht zielführend, mehr als 2 Eisbuden in diesem Intervall zu platzieren.

1.1.3 Eine Eisbude

Wird eine Eisbude in diesem Intervall bei Position $k \in I$ platziert, soll nun betrachtet werden, welche Häuser für den Vorschlag stimmen werden. Steht ein Haus bei Position $x \leq k$ und war zuvor die Eisbude bei 0 näher für dieses, so war der alte Abstand x, womit der neue maximal $x-1$ sein darf. Da der Abstand beim neuen Vorschlag $k - x$ ist, wird der Bewohner bei x genau dann für den Vorschlag stimmen, wenn

$$k - x \leq x - 1 \tag{1}$$
$$k + 1 \leq 2x \tag{2}$$
$$\frac{k+1}{2} \leq x \tag{3}$$

Für ein Haus bei $y \geq k$, für das vorher bei m die nächste Eisbude war, muss analog gelten

$$y - k \leq m - y - 1 \tag{4}$$
$$2y \leq m + k - 1 \tag{5}$$
$$y \leq \frac{m+k-1}{2} \tag{6}$$

Da für x Position 0 näher sein muss und für y Position m, muss ferner

$$x \leq \frac{m}{2} \qquad\qquad y \geq \frac{m}{2} \tag{7}$$

Sei x_{min} das minimale x, das dafür stimmt und y_{max} das maximale y. Dann werden alle Häuser in $A = [x_{min}; \frac{m}{2}]$ und $B = [\frac{m}{2}; y_{max}]$, also alle in $J = [x_{min}; y_{max}]$ dafür stimmen. Da x und y ganzzahlig, folgt aus (3) bzw. (6):

$$x_{min} = \left\lceil \frac{k+1}{2} \right\rceil \qquad\qquad y_{max} = \left\lfloor \frac{m+k-1}{2} \right\rfloor \tag{8}$$

Bei dieser Formel fällt auf, dass bei einem Erhöhen von k um 2 das Intervall J um eine Einheit nach rechts „verschoben“ wird:

$$x_{min,k+2} = \left\lceil \frac{(k+2)+1}{2} \right\rceil = \left\lceil \frac{k+1}{2} + 1 \right\rceil = \left\lceil \frac{k+1}{2} \right\rceil + 1 \qquad = x_{min} + 1 \tag{9}$$
$$y_{max,k+2} = \left\lfloor \frac{m+(k+2)-1}{2} \right\rfloor = \left\lfloor \frac{m+k-1}{2} + 1 \right\rfloor = \left\lfloor \frac{m+k-1}{2} \right\rfloor + 1 \qquad = y_{max} + 1 \tag{10}$$

Es gibt also 2 „Startintervalle“, eines für gerades und eines für ungerades k, wonach die anderen Intervalle jeweils durch Verschiebung auseinander hervorgehen. Es fällt auf, dass immer ein ungerades $k = 2v + 1$ anstelle eines geraden $k = 2v$ gewählt werden kann, denn

$$x_{min,ungerade} = \left\lceil \frac{2v+2}{2} \right\rceil = \left\lceil \frac{2v+1}{2} \right\rceil = x_{min,gerade} \tag{11}$$

Es sind also beide x_{min} Werte gleich, aber bei der ungeraden Version ist k größer, womit $y_{max,ungerade} \geq y_{max,gerade}$. Somit ist bei gleichem Intervallanfang das Ende bei einem Wert, der nicht kleiner ist, falls k ungerade gewählt wird. Es ist also nur eine Intervallgröße zu betrachten, durch die dann mittels Verschiebung alle relevanten Intervalle zustande kommen. Deshalb soll nun noch die Intervallgröße g bestimmt

werden:

$$g = y_{max} - x_{min} = \left\lfloor \frac{m+k-1}{2} \right\rfloor - \left\lceil \frac{k+1}{2} \right\rceil \tag{12}$$

$$= \left\lfloor \frac{m+(2v+1)-1}{2} \right\rfloor - \left\lceil \frac{(2v+1)+1}{2} \right\rceil \tag{13}$$

$$= \left\lfloor \frac{m+2v}{2} \right\rfloor - \left\lceil \frac{2v+2}{2} \right\rceil \tag{14}$$

$$= \left\lfloor \frac{m}{2} + v \right\rfloor - \lceil v+1 \rceil \tag{15}$$

$$= \left\lfloor \frac{m}{2} \right\rfloor + v - (v+1) \tag{16}$$

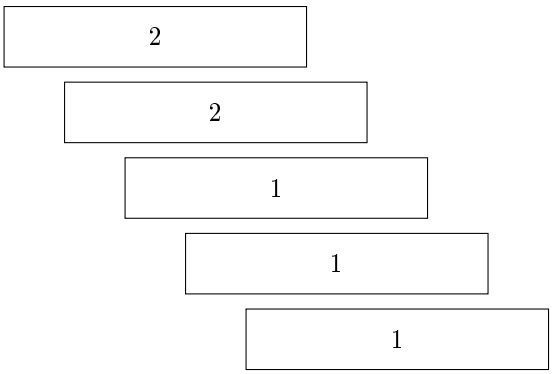
$$= \left\lfloor \frac{m}{2} \right\rfloor - 1 \tag{17}$$

Wird also eine Eisbude in I platziert, so ist die maximale Anzahl an gewonnen Stimmen in diesem Intervall gleich der maximalen Intervallsumme:

$$\max_{1 \le a \le m-1-g} \left(\sum_{i=a}^{a+g} v_i \right) \tag{18}$$

Dabei gibt v_i die Stimmenanzahl bei Position i an. Konkret ist $v_i = 1$, wenn sich bei i ein Haus befindet und sonst 0. Die Anzahl an Stimmen, die so mit einer Eisbude geholt werden kann, wird im Folgenden durch mx angegeben.
Zur Veranschaulichung wird ein Beispiel mit $m = 10$ betrachtet. Es gilt $g = \frac{10}{2} - 1 = 4$. Somit können in allen Intervallen der Größe 4 jeweils alle Stimmen mit einer Eisbude geholt werden. Diese Stimmenanzahl bzw. Intervallsumme wird im Folgenden dargestellt:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
H		H	H					H				H		H	H				
E										E									



Die Größte Intervallsumme ist 2, womit $mx = 2$.

1.2 Vorberechnungen

1.2.1 Präfixsummen

Es ist wichtig schnell herausfinden zu können, wie viele Häuser sich in einem bestimmten Intervall befinden. Die Anzahl der Häuser bzw. Stimmen in einem Intervall verändert sich nicht, weshalb diese über vorberechnete Präfixsummen effizient berechnet werden können. Ist nämlich

$$p_i = \sum_{j=0}^{i} v_j \tag{19}$$

so ist

$$\sum_{j=a}^{b} v_j = \sum_{j=0}^{b} v_j - \sum_{j=0}^{a-1} v_j = p_b - p_{a-1} \tag{20}$$

Die $\mathcal{O}(u)$ Präfixe können jeweils in $\mathcal{O}(1)$ aus dem vorherigen Wert berechnet werden, denn

$$p_{i+1} = \sum_{j=0}^{i+1} v_j = v_{i+1} + \sum_{j=0}^{i} v_j = v_{i+1} + p_i \tag{21}$$

Insgesamt ist dieser Schritt der Vorberechnung also in $\mathcal{O}(u)$ möglich, wonach die Summe von Intervallen in konstanter Zeit berechnet werden kann. Da das Array aber zyklisch ist (der See ist ein Kreis), muss gesondert mit dem Fall umgegangen werden, bei dem ein Intervall von a nach b gefragt ist, mit $a > b$. In diesem Fall muss das Intervall in 2 Teile zerlegt werden. Einer von a bis zum letzten Element (bei $u-1$) und einer von 0 bis b.

1.2.2 Maximale Intervalle

Es gibt $\mathcal{O}(u^2)$ Möglichkeiten, 2 Eisbuden zu platzieren. Für alle wird mx vorberechnet, was also insgesamt $\mathcal{O}(u^3)$ Arbeit bedeutet, denn es müssen jeweils $\mathcal{O}(u)$ Intervalle getestet werden, deren Summen nach 1.2.1 je in $\mathcal{O}(1)$ bestimmt werden können.

1.2.3 Nutzen der Vorberechnungen

Um nun zu überprüfen, ob eine Situation stabil ist, müssen folgende Fälle für die Platzierung der 3 Eisbuden getestet werden:

- Alle neuen Eisbuden werden je zwischen zwei unterschiedlichen Eisbuden platziert. Dann ist die Anzahl der erhaltenen Stimmen die Summe der mx.
- Zwei neue Eisbuden werden zwischen denselben Eisbuden platziert. Dann ist nach 1.1.2 die Anzahl der erhaltenen Stimmen die Summe des gesamten Intervalls sum. Die dritte Eisbude wird zwischen zwei andere Eisbuden platziert und holt mx Stimmen.

Beide Fälle sind nach den Vorberechnungen in konstanter Zeit auswertbar. Da es nur sehr wenige Kombinationen für diese Fälle gibt (Bei Fall 1 gibt es nur eine, bei Fall 2 6 also insgesamt 7), können diese per Brute-Force getestet werden[1].

[1] Bei variabler Eisbudenanzahl sollte dies jedoch nicht mit Brute-Force gemacht werden. Stattdessen ist es möglich dieses Problem auf einen Spezialfall des Rucksackproblems zu reduzieren, welcher über einen Greedy-Ansatz sehr effizient gelöst werden kann: Alle Objekte haben Gewicht 1. Es werden Objekte mit Werten mx und $sum - mx$ eingefügt. Da alle Objekte dasselbe Gewicht haben, können gierig die Objekte nach Wert sortiert genommen werden. Dies wäre bei 3 Eisbuden in der Praxis jedoch wahrscheinlich kaum schneller.

1.3 Beispiel zur Veranschaulichung

Um besser verstehen zu können, wie überprüft wird, ob eine Eisbudenverteilung stabil ist, soll eine Situation aus Beispieleingabe 1 betrachtet werden:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
H		H	H					H				H		H	H				
		E			E							E							

Für die Präfixsumme ergibt sich:

1	1	2	3	3	3	3	3	4	4	4	4	5	5	6	7	7	7	7	7

Um nun z.B. festzustellen, wie viele Häuser zwischen der dritten und ersten Eisbude liegen, wird das zyklische Array in zwei Teile geteilt. Einer von Index 13 bis 19 und einer von Index 0 bis 1. Über die Präfixsummen lässt sich nun effizient die Anzahl der Häuser in diesem Intervall bestimmen:

$$\sum_{i=13}^{19} v_i = p_{19} - p_{12} = 7 - 5 = 2$$

$$\sum_{i=0}^{1} v_i = p_1 = 1$$

Aus $1 + 2 = 3$ folgt, dass sich zwischen diesen beiden Eisbuden 3 Häuser befinden. Der Abstand zwischen diesen beiden Eisbuden beträgt 10 LE, da $2 - 12 \equiv -10 \equiv 20 - 10 \equiv 10 \pmod{20}$. Die maximale Stimmenanzahl mx, die eine Eisbude bei einem neuen Vorschlag in diesem Intervall holen kann, ist also die maximale Arraysumme in einem Intervall der Größe $g = \left\lfloor \frac{10}{2} \right\rfloor - 1 = 4$. Dieses kann nur 2 Häuser umfassen. Hier ist ein mögliches Intervall mit maximaler Summe dargestellt:

													–	H	H	–	–		

Man kann also zwischen der dritten und ersten Eisbude durch das platzieren von einer Eisbude maximal zwei Stimmen holen und mit zwei Eisbuden alle drei, die sich dazwischen befinden. Analog können für die beiden anderen Intervalle (Eisbude 1 (Index 2) - Eisbude 2 (Index 5) und Eisbude 2 - Eisbude 3) die Werte sum und mx bestimmt werden. Insgesamt erhält man folgendes Ergebnis:

	E1 - E2	E2 - E3	E3 - E1
Stimmen mit einer Eisbude (mx)	1	1	2
Stimmen mit zwei Eisbuden (sum)	1	1	3

Optimal werden die drei Eisbuden so verteilt, dass sich in jedem Intervall eine oder im letzten Intervall zwei befinden. In beiden Fällen hat der neue Vorschlag $1 + 1 + 2 = 4$ bzw. $1 + 3 = 4$ Stimmen, was bei 7 Häusern die Mehrheit ist. Damit ist diese Verteilung nicht stabil. Analog wird für alle anderen Eisbudenverteilungen vorgegangen, bis eine stabile Verteilung gefunden wird oder sich herausstellt, dass so eine nicht existiert.

1.4 Optimierungen

Der bisherige Ansatz hat eine Laufzeit von $\mathcal{O}(u^3)$. Damit lassen sich die Beispiele der BWINF-Webseiten bereits lösen. Das größte Beispiel ($u = 625$) kann in wenigen Sekunden gelöst werden. Schwierig wird es aber bei größeren Beispielen mit $u = 10000$. Das Programm soll in der Praxis möglichst schnell sein, weshalb der Ansatz nun soweit optimiert werden soll, dass am Ende auch solche Beispiele lösbar sind. Die Laufzeit wird dabei auf $\mathcal{O}(u^2)$ reduziert.

1.4.1 Vorberechnung mit Monotonie-Trick

Da bereits die Vorberechnung $\mathcal{O}(u^3)$ Zeit benötigt, muss sich hieran etwas ändern. Ein möglicher Ansatz wäre das Berechnen der mx, wenn sie benötigt werden und anschließende Speicherung. Es ist aber schwer abzuschätzen, wie viele mx Werte benötigt werden. Es könnten im schlimmsten Fall auch $\mathcal{O}(u^2)$ sein, was mit einer Berechnungsdauer von $\mathcal{O}(u)$ keinen Vorteil liefern würde. Stattdessen soll die Geschwindigkeit

der Vorberechnungen erhöht werden.
Betrachtet wird die Berechnung von mx zwischen zwei Eisbuden A und B. Werden beide um eine Einheit nach rechts verschoben, so können viele Intervallsummen weiterhin betrachtet werden, da sich der Abstand zwischen den beiden Eisbuden nicht ändert und somit auch g konstant bleibt. Ein Intervall fällt dann aus dem aktuellen Bereich raus (Dieses hat bei A begonnen) und ein neues kommt hinzu (Dieses endet direkt vor B):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
H		H	H					H				H		H	H				
		A	**A**												B	**B**			

2 **3**

Es fällt also immer genau ein Wert aus dem Bereich heraus und ein neuer kommt hinzu, wobei die Werte die Summen der Intervalle sind. Es ist also für jedes „Fenster“ das Maximum der Werte in diesem zu bestimmen. Zur besseren Übersichtlichkeit wird dieses Problem auf einem Array betrachtet, da so die einzelnen Werte leichter zu erkennen sind. Als Beispiel wird dieses Array mit Fenstergröße 4 betrachtet:

$$\underbrace{1\ 4\ 3\ 2}_{4}\ 3\ 5\ 2$$
$$1\ \underbrace{4\ 3\ 2\ 3}_{4}\ 5\ 2$$
$$1\ 4\ \underbrace{3\ 2\ 3\ 5}_{5}\ 2$$
$$1\ 4\ 3\ \underbrace{2\ 3\ 5\ 2}_{5}$$

Hierbei fällt auf, dass die Reihenfolge der Indizes, die zu den maximalen Elementen gehören, monoton steigend ist. So kann es nicht sein, dass sich erst bei Index 3 das größte Element befindet und nach dem Verschieben des Fensters nach rechts das größte Element bei Index 2 ist. Sonst wäre ja zuvor bereits das größte Element bei Index 2 gewesen, da es bereits im Fenster vorhanden war.
Die Idee ist nun, eine absteigend sortierte Liste an möglichen Kandidaten für das größte Element zu verwalten. Wenn am rechten Fensterrand eine Zahl x dazu kommt, werden am Ende der Liste alle Zahlen entfernt, die kleiner als x sind und anschließend wird x zur Liste hinzugefügt. Dies funktioniert durch die monotone Reihenfolge der optimalen Indizes: Da x größer ist, als eine Zahl y, die aus der Liste entfernt wird, kann y aktuell nicht optimal sein. Aber auch später kann y nicht das Maximum sein, da x nach y hinzugefügt wurde und somit länger im Fenster bleibt.
Dadurch, dass die Liste absteigend sortiert ist, steht das Maximum am Anfang der Liste. Falls dieses am linken Rand aus dem Fenster verschwindet, muss es aus der Liste entfernt werden. Nun sollen die Elemente der Liste am vorherigen Beispiel veranschaulicht werden:

$$\underbrace{1\ 4\ 3\ 2}_{4\ 3\ 2}\ 3\ 5\ 2$$
$$1\ \underbrace{4\ 3\ 2\ 3}_{4\ 3}\ 5\ 2$$
$$1\ 4\ \underbrace{3\ 2\ 3\ 5}_{5}\ 2$$
$$1\ 4\ 3\ \underbrace{2\ 3\ 5\ 2}_{5\ 2}$$

- Die anfänglichen Zahlen 1, 4, 3, 2 werden der Reihe nach zur Liste hinzugefügt. Da die 1 vor der größeren 4 kommt, wird sie entfernt.
- In Schritt 2 wird die 3 hinzugefügt. Alle Element am Ende der Liste, die kleiner gleich 3 sind (also die 2 und 3) werden entfernt, danach wird die 3 hinzugefügt. Die 4 ist weiterhin links und damit das größte Element.

- In Schritt 3 fällt zunächst das Maximum 4 links aus dem Intervall. Dafür kommt die 5 dazu, die größer als die verbleibende 3 ist, weshalb die 3 auch entfernt wird.
- In Schritt 4 wird die 2 hinzugefügt.

Im Pseudocode wurde der erste Schritt, also die anfängliche Befüllung der Liste weggelassen, da dieser genau wie der Rest funktioniert, nur ohne das Verändern der linken Grenze. Das Array a enthält die verbleibenden Elemente, also alle, die noch nicht im Fenster sind:

Algorithm 1 Maximum in Fenster mit Monotonie-Trick

```
a ← Array mit den verbleibenden Zahlen
list ← Liste mit den anfänglichen Zahlen
for all i in 1...n do
    if Erstes Element von list nicht mehr im Fenster then
        list.remove_first_element()
    while a[i] ≥ list.last_element() do
        list.remove_last_element()
    list.append(a[i])
```

Dieses Vorgehen, um die Maxima jedes Fensters zu bestimmen, hat nur eine lineare Laufzeit. Dies lässt sich damit Begründen, dass jedes Element genau einmal zur Liste hinzugefügt wird und höchstens einmal aus ihr entfernt wird. Es wird also für das Bestimmen aller mx Werte für einen Eisbudenabstand von m nur noch $\mathcal{O}(u)$ Zeit benötigt. Da $\mathcal{O}(u)$ Werte von m existieren, für die das gemacht werden muss, kommt man am Ende auf eine neue Laufzeit von $\mathcal{O}(u^2)$, was bereits eine deutliche Verbesserung zum anfänglichen $\mathcal{O}(u^3)$ Ansatz ist.

1.4.2 Eine der Eisbuden steht direkt bei einem Haus

Da nun die Vorberechnung effizient möglich ist, muss noch das Suchen der Lösung effizienter gestaltet werden. Es stellt sich heraus, dass falls eine Lösung existiert, auch eine solche existiert, bei der sich eine Eisbude direkt bei einem Haus befindet. Dieser Satz soll nun konstruktiv bewiesen werden:

Der Fall, bei dem bereits eine oder mehr Eisbuden bei einer stabilen Verteilung direkt bei einem Haus stehen, ist uninteressant, denn in diesem Fall ist der Satz wahr. Also wird angenommen, dass eine stabile Verteilung existiert, bei der sich keine Eisbude direkt bei einem Haus befindet. Nun können alle drei Eisbuden in dieselbe Richtung bewegt werden, bis eine der Eisbuden nur noch eine Einheit von einem Haus entfernt ist. Die Intervallsummen, die dabei entfernt und hinzugefügt werden, ändern nichts am Maximum mx zwischen je zwei Eisbuden, denn das maximale Intervall kann immer bei einem Haus beginnen (Abb. 1) und es bleiben dieselben Häuser in den Intervallen (Die Eisbuden werden ja nur bewegt bis eine direkt neben einem Haus steht). In dieser Situation können die Eisbuden noch um eine Einheit weiter bewegt werden. Das Maximum kann hier ebenfalls nicht größer werden, denn es werden keine neuen Häuser zu den Intervallen hinzugefügt, es werden sogar weniger (ein Haus würde sich ja jetzt bei einer Eisbude befinden und somit nicht mehr in einem der Intervalle), womit die Verteilung stabil bleibt. Es lässt sich also durch Verschiebung aller Eisbuden immer aus einer Lösung ohne Eisbude bei einem Haus eine mit einer solchen Eisbude konstruieren.

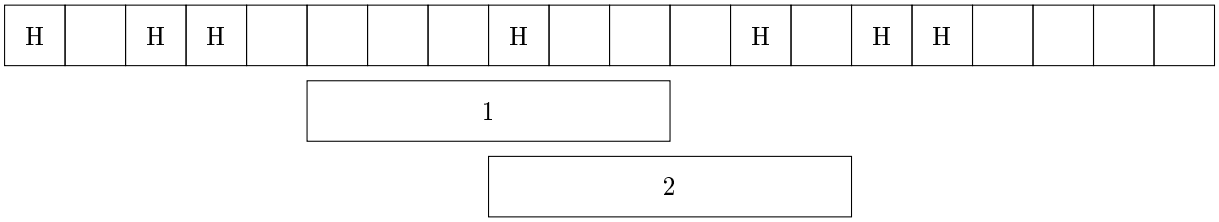


Abbildung 1: Leere Plätze links am Intervall bringen keinen Vorteil! Das Intervall kann direkt beim Haus starten und es sind nicht weniger Häuser in dem Intervall.

1.4.3 Frühzeitiges Abbrechen der Suche

Bei dem aktuellen Ansatz wird die Stabilität erst überprüft, nachdem die drei Eisbuden platziert wurden. Es kann aber vorkommen, dass die Position der ersten beiden Eisbuden bereits eine stabile Verteilung unmöglich macht. Hierfür wird angenommen, dass sich zwischen zwei Eisbuden x Häuser befinden. Es können also mit zwei Eisbuden x Stimmen geholt werden. Die übrige Eisbude wird dann in einem der 2 verbleibenden Intervalle platziert und kann dort mindestens ein Viertel der verbleibenden Stimmen holen[2]. Selbst wenn alle Eisbuden bei einem Haus stehen, bleiben mindestens $n-3-x$ Stimmen übrig. Es lassen sich also mindestens

$$x + \frac{n-3-x}{4} \tag{22}$$

Stimmen holen, wenn sich x Häuser zwischen zwei Eisbuden befinden. Nun soll bestimmt werden, für welche Werte von x das bereits einer Mehrheit entspricht, welche die Verteilung instabil macht:

$$x + \frac{n-3-x}{4} > \frac{n}{2} \tag{23}$$

$$4x + n - 3 - x > 2n \tag{24}$$

$$3x > n + 3 \tag{25}$$

$$x > \frac{n}{3} + 1 \tag{26}$$

Sobald sich also mehr als $\frac{n}{3}+1$ Häuser zwischen zwei Eisbuden befinden, ist die Verteilung instabil, unabhängig davon, wie die dritte platziert wird.
Es lässt aber auch eine minimale Anzahl an Häusern bestimmen. Wenn nämlich $\frac{n}{3}-6$ oder weniger Häuser zwischen zwei Eisbuden liegen, müssen in einem der beiden anderen Intervalle mehr als $\frac{n}{3}+1$ Häuser liegen. Denn auf diese werden mindestens $n-3-(\frac{n}{3}-6) = \frac{2}{3}n+3$ Häuser verteilt. Nach dem Schubfachprinzip muss eines der beiden Intervalle also mindestens

$$\left\lceil \frac{\frac{2}{3}n+3}{2} \right\rceil = \left\lceil \frac{n}{3} + \frac{3}{2} \right\rceil \geq \frac{n}{3} + \frac{3}{2} > \frac{n}{3} + 1 \tag{27}$$

Häuser enthalten, womit die Verteilung instabil ist. Damit also überhaupt die Möglichkeit einer stabilen Verteilung besteht, muss für die Anzahl an Häusern x zwischen zwei Eisbuden gelten:

$$\frac{n}{3} - 6 < x \leq \frac{n}{3} + 1 \tag{28}$$

$$\left\lfloor \frac{n}{3} \right\rfloor - 5 \leq x \leq \left\lfloor \frac{n}{3} \right\rfloor + 1 \tag{29}$$

Der letzte Schritt folgt daraus, dass x als Anzahl von Häusern ganzzahlig sein muss.
Wird also zuerst eine Eisbude bei einer beliebigen Position platziert, kann die zweite nur bei einem Haus positioniert werden, welches zwischen sich und der bereits gewählten Eisbude $\lfloor \frac{n}{3} \rfloor + x$ Häuser hat mit $x \in [-5;1]$. Dies funktioniert, da nach 1.4.2 im Fall einer stabilen Verteilung immer eine solche existiert, bei der sich eine Eisbude (hier die zweite) bei einem Haus befindet. Für die Wahl der zweiten Eisbude gibt es also nur $1-(-5)+1 = 7 = \mathcal{O}(1)$ Möglichkeiten, womit für die Wahl der ersten zwei Eisbuden nur noch $\mathcal{O}(u)$ Möglichkeiten bleiben. Für alle drei sind es dann $\mathcal{O}(u^2)$. Da für die Vorberechnung auch nur $\mathcal{O}(u^2)$ Zeit benötigt wird, ist dies auch die Gesamtlaufzeit der Lösung.

2 Unabhängigkeit vom Umfang

Der bisherige Ansatz kann zwar ohne Probleme die Beispiele der BWINF-Webseiten lösen, aber die Laufzeit ist stark abhängig vom Umfang des Sees. Wenn also an einem großen See nur wenige Häuser stehen, wartet man eventuell lange auf das Ergebnis... Auch mit nichtganzzahligen Adressen kommt der Ansatz noch nicht klar, da in diesem Fall theoretisch unendlich viele Positionen für die Eisbuden existieren.

[2] Es gibt zwei Paare von Eisbuden zwischen denen sich noch keine neue befindet. Man kann das mit den meisten Häuser wählen. Platziert man nun eine Eisbude am linken oder rechten Rand dieses Intervalls, holt man bei einer der beiden Varianten mindestens die Hälfte der Stimmen in diesem Intervall, da man so die linke oder rechte Hälfte des Intervalls für den Vorschlag gewinnen kann. Man kann also zweimal hintereinander mindestens die Hälfte der Stimmen holen, also insgesamt mindestens ein Viertel.

2.1 Überblick

Es werden nicht alle Positionen der Eisbuden durchprobiert, sondern nur das Haus, bei dem sich eine Eisbude befindet, oder die beiden Häuser, zwischen denen sich die Eisbude befindet. Dadurch ist die Laufzeit für das Probieren unabhängig von u und nur noch abhängig von n. Da so für alle Eisbuden vorgegeben ist, welche Häuser „vor“ bzw. „nach“ ihnen kommen, ist die Anzahl sum an Häusern zwischen je zwei Eisbuden nach diesem Schritt bereits eindeutig festgelegt. Nur die maximale Stimmenzahl mx, die mit einer Eisbude geholt werden kann, kann noch verändert werden, falls sich eine Eisbude zwischen zwei Häusern befindet. Die Idee ist nun, die möglichen Werte von mx durchzuprobieren. Falls welche gefunden werden, die zu einer stabilen Verteilung führen, müssen nun die Positionen der Eisbuden genauer bestimmt werden.

2.2 Maximale Intervallgröße

Die Idee ist, für ein Intervall von Häusern den maximalen Abstand $size$ zwischen zwei Eisbuden zu bestimmen für die der Wert von mx eine bestimmte Größe mx' nicht überschreitet. Wenn dann also in dem Intervall zwei Eisbuden mit Abstand $\leq size$ platziert werden, ist $mx \leq mx'$.
Um $size$ zu bestimmen, wird die kleinste Intervallgröße G bestimmt, bei der $mx > mx'$ zu groß ist. Dann gilt $size = 2G + 1$, denn der tatsächliche Wert von g ist dann kleiner als G und somit nicht zu groß:

$$g = \left\lfloor \frac{2G+1}{2} \right\rfloor - 1 = G - 1 < G \tag{30}$$

So ist $size$ aber auch maximal, denn der um eins größere Wert führt zu $g = G \implies mx > mx'$:

$$g = \left\lfloor \frac{2G+2}{2} \right\rfloor - 1 = G \tag{31}$$

Zur Bestimmung von G muss das kleinste Intervall gefunden werden, dass $mx' + 1$ Häuser enthält. Dieses kleinste Intervall startet und endet offensichtlich bei einem Haus, denn sonst könnte man es an den Enden verkleinern. Als Beispiel wird eine Situation mit $mx' = 2$ betrachtet:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
		H	H					H				H		H	H				

Das oben abgebildete Intervall ist das kleinste, dass $mx' + 1 = 3$ Häuser beinhaltet. Das untere demonstriert, dass am linken und rechten Ende freie Stellen ohne Haus entfernt werden können, weshalb es nicht das kleinste Intervall sein kann. In dem Beispiel ist also $G = 3$, da die Indizes des ersten und letzten Hauses im Intervalls eine Differenz von 3 besitzen. Daraus folgt $size = 2G + 1 = 7$. Wenn hier also zwei Eisbuden mit Abstand ≤ 7 platziert werden, ist $mx \leq mx'$, da ansonsten G nicht minimal wäre.
Zur Bestimmung von G muss von allen betrachteten Häusern mit Indexdifferenz mx' der minimale Abstand gefunden werden. Im obigen Beispiel wären diese Abstände:

$$8 - 2 = 6 \qquad 12 - 3 = 9 \qquad 14 - 8 = 6 \qquad 15 - 12 = 3$$

Wenn die Werte von G bzw. $size$ nacheinander vorberechnet werden, wobei mx gleich bleibt, kann hier erneut der Monotonie-Trick verwendet werden: Es kommt ein Haus am rechten Rand in das Intervall und eines verschwindet am linken Rand. Für das rechte Haus kommt ein neuer Abstand bzw. Kandidat für G dazu und der Abstand, der bei dem linken Haus begonnen hat, fällt nun aus dem betrachteten Fenster. Anders als vorher wird aber nicht mehr das Maximum, sondern das Minimum gesucht. Daher ist die Liste nun aufsteigend sortiert und hintere Elemente werden entfernt, falls sie größer als das neue Element sind.

2.3 Finden der genauen Position

Nachdem nun die Werte von $size$ bekannt sind, also die maximalen Abstände, die die Eisbuden haben dürfen, müssen die Positionen der Eisbuden bestimmt werden, die zwischen zwei Häusern liegen. Als

Vereinfachung wird angenommen, dass sich die erste Eisbude direkt bei einem Haus befindet. Dies ist möglich, da im Falle einer Lösung auch eine solche existiert, bei der sich eine der Eisbuden direkt bei einem Haus befindet (1.4.2). Es ist also die Position von Eisbude 1 bekannt. Falls die der nächsten auch schon bekannt ist, kann sie einfach dort platziert werden. Ansonsten sollte der Abstand von der 2. zur 1. so groß wie möglich gewählt werden. Denn solange $size$ nicht überschritten wird, bleibt mx noch in einer Größe, die zu einer stabilen Verteilung führt. Somit schadet es nicht, den Abstand so groß wie möglich zu wählen. Im Gegenteil: Dadurch werden die anderen Abstände kleiner, da alle zusammen u ergeben. Ein Abstand, der nicht so groß wie möglich ist, könnte hingegen bewirken, dass einer der anderen Abstände größer als $size$ sein muss. Somit ist es immer optimal die Abstände so groß wie möglich zu wählen. Dabei muss aber berücksichtigt werden, dass die Eisbude weiterhin zwischen denselben Häusern stehen muss. Die Position der 2. Eisbude ist dann also $\min(a + size, h - 1)$, wobei h das nächste Haus ist und a die erste Eisbude. Wenn dann die Position der zweiten Eisbude bekannt ist, kann die dritte nach demselben Verfahren platziert werden, da die zweite nun eine bekannte Position hat.
Nachdem alle Eisbude platziert wurden, ist zu kontrollieren, ob sie jeweils Abstände kleiner gleich $size$ haben. In diesem Fall wurde eine stabile Verteilung gefunden. Ansonsten lassen sich die Eisbuden nicht mit den Abständen platzieren, da der beschriebene Greedyansatz sonst eine Lösung gefunden hätte. Dann muss weiter gesucht werden.

2.4 Veranschaulichung

Abbildung 2 stellt eine Situation aus eisbuden1.txt dar:

1. Das Beispiel (vgl. eisbuden1.txt). Die Häuser sind die grünen Rechtecke.
2. Die Eisbuden (violette Kreise) werden entweder bei einem Haus oder in einem Zwischenraum platziert. Die zweite Eisbude befindet sich zwischen zwei Häusern bei 8 und 12. Ihre genaue Position ist aber noch unbekannt.
3. Nun werden die Werte für mx durchprobiert. Hier $mx = 1$ zwischen je zwei Eisbuden.
4. Berechnung des maximalen Abstandes $size$ zwischen je zwei Eisbuden, sodass der mx Wert nicht größer wird als der, der gerade probiert wird.
   $size$ zwischen der ersten und zweiten Eisbude ist z.B. 11. In diesem Fall ist $g = \lfloor \frac{11}{2} \rfloor - 1 = 4$. Offensichtlich können die Häuser bei 3 und 8 nicht in demselben Intervall liegen, da sie einen Abstand von 5 haben. Wäre $size = 12$, wäre $g = 5$, womit beide in einem Intervall liegen würden. Dann wäre aber $mx = 2$. Somit ist $size = 11$ der größtmögliche Abstand zwischen den ersten beiden Eisbuden, wenn $mx = 1$.
   Zwischen den Eisbuden 2 und 3 ist $size = \infty$ angegeben, da für die Berechnung von $size$ nur Häuser betrachtet werden, die sich zwischen den Eisbuden befinden. Hier ist es nur ein Haus. Unabhängig davon, wie groß das Intervall gewählt wird, kann also nur ein Haus in ihm liegen.
5. Versuch, die Eisbuden so zu platzieren, dass Abstände nie größer als $size$ werden. Hier muss nur eine Position für Eisbude 2 gesucht werden, da die anderen bei einem Haus stehen. Diese wird bei 11 platziert, da die Abstände bei dem Greedyansatz immer so groß wie möglich gewählt werden, ohne dass sich die Häuser ändern, zwischen denen die Eisbude steht.

2.5 Lösung in Linearzeit

Nachdem die erste Eisbude bei einem Haus platziert wurde, gibt es nach 1.4.3 nur konstant viele Möglichkeiten, die zweite Eisbude zu platzieren. Danach gibt es erneut nur konstant viele Möglichkeiten für die dritte Eisbude. Somit gibt es nur $\mathcal{O}(n)$ Eisbudenverteilungen, die durchprobiert werden müssen.

Bei mx müssen tatsächlich auch nur konstant viele Werte durchprobiert werden. Hierzu wird die folgende Beobachtung benötigt: $mx \geq \frac{sum}{2}$. Mit einer Eisbude lässt sich immer mindestens der Hälfte der Stimmen holen. Wird die Eisbude ganz links platziert, wird die linke Hälfte der Häuser dafür stimmen. Wird sie rechts platziert, erhält man die rechte Hälfte der Stimmen. Man kann also schreiben $mx = \lceil \frac{sum}{2} \rceil + \Delta$ mit einem $\Delta \in \mathbb{N}_0$. Es müssen für drei Intervalle Werte für mx probiert werden. Also müssen $\Delta_1, \Delta_2, \Delta_3$

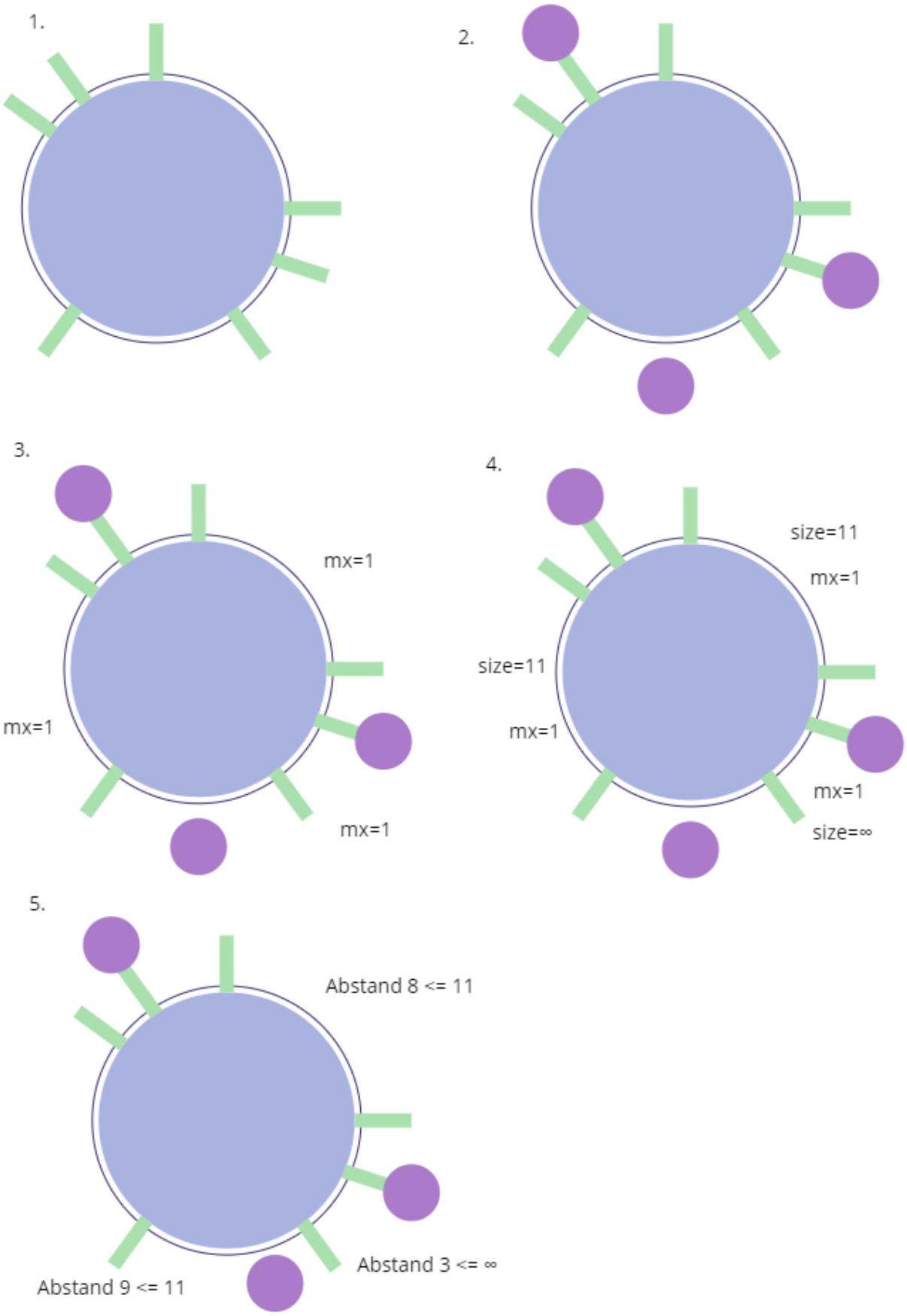


Abbildung 2: Veranschaulichung der dritten Lösungsidee

durchprobiert werden. Wenn aber $\Delta_1 + \Delta_2 + \Delta_3 \geq 2$, ist die Summe der mx_1:

$$mx_1 + mx_2 + mx_3 = \left\lceil \frac{sum_1}{2} \right\rceil + \Delta_1 + \left\lceil \frac{sum_2}{2} \right\rceil + \Delta_2 + \left\lceil \frac{sum_3}{2} \right\rceil + \Delta_3 \tag{32}$$

$$\geq \frac{sum_1 + sum_2 + sum_3}{2} + \Delta_1 + \Delta_2 + \Delta_3 \tag{33}$$

$$\geq \frac{n-3}{2} + 2 \tag{34}$$

$$> \frac{n}{2} \tag{35}$$

Somit ist die Anzahl der benötigten Überprüfungen die Anzahl an Möglichkeiten drei Δ Werte mit einer Summe < 2 zu wählen. Das sind insgesamt $4 = \mathcal{O}(1)$ Möglichkeiten[3]. Es müssen also insgesamt nur $\mathcal{O}(n)$ Möglichkeiten durchprobiert werden.

Bei der Vorberechnung können durch den Monotonie-Trick $\mathcal{O}(n)$ *size* Werte mit gleichem mx' in $\mathcal{O}(n)$ auf einmal berechnet werden. Da nur $\mathcal{O}(1)$ relevante mx bzw. mx' Werte existieren, kann der Schritt der Vorberechnung ebenfalls in $\mathcal{O}(n)$ ablaufen, womit die gesamte Lösung nur eine lineare Laufzeit hat.
Dies ist auch die theoretisch beste erreichbare Laufzeit, da bereits die Eingabe durch die n Hauspositionen eine Größe von $\mathcal{O}(n)$ hat. Somit benötigt jedes Programm $\Omega(n)$ Zeit, da es die Eingebe lesen muss. Allerdings setzt diese Laufzeit voraus, dass die Hauspositionen in sortierter Reihenfolge sind. Ansonsten müssten diese zunächst sortiert werden, was wiederum zu einer Laufzeit von $\mathcal{O}(n \log(n))$ führen würde.[4]

3 Erweiterung

Als Erweiterung soll das Programm auch mit nichtganzzahligen Positionen umgehen können. Dafür kann der bisherige Ansatz in den Grundzügen weiter verwendet werden. Allerdings muss die Intervallgröße g neu hergeleitet werden.
Für das Haus x musste vorher gelten

$$k - x \leq x - 1 \tag{36}$$

da x der bisherige Abstand zur nächsten Eisbude bei 0 war und bei der Betrachtung ganzer Zahlen $x-1$ somit der maximale neue Abstand sein durfte. Durch die Betrachtung reeller Zahlen, reicht nun aber:

$$k - x < x \tag{37}$$

$$\frac{k}{2} < x \tag{38}$$

Analog für y:

$$y - k < m - y \tag{39}$$

$$y < \frac{m+k}{2} \tag{40}$$

Für die Differenz der minimalen x- bzw. maximalen y-Werte gilt:

$$g = y_{max} - x_{min} \tag{41}$$

$$= \frac{m+k}{2} - \frac{k}{2} - \varepsilon \tag{42}$$

$$= \frac{m}{2} - \varepsilon \tag{43}$$

wobei ε eine beliebig kleine Zahl größer 0 ist, die hier benötigt wird, da z.B. y nicht genau $\frac{m+k}{2}$ sein darf, sondern ein bisschen kleiner sein muss.
Für die Berechnung von *size* gilt nun $size = 2G$, denn $g = \frac{2G}{2} - \varepsilon = G - \varepsilon < G$.
Abgesehen davon, kann weiterhin zunächst der Zwischenraum von zwei Häusern probiert werden, in dem sich eine Eisbude befindet. Dann werden die vorberechneten *size* Werte verwendet, um eine konkrete Positionierung der Eisbuden zu finden. Diese Schritte funktionieren ebenfalls mit nichtganzzahligen Werten. Somit beträgt die Laufzeit weiterhin $\mathcal{O}(n)$ bzw. $\mathcal{O}(n \log(n))$.

[3] Entweder alle sind 0 oder einer der Werte ist 1.

[4] Genaugenommen kann die Laufzeit auch mit unsortierten Werten bei $\mathcal{O}(n)$ liegen, da die $\Omega(n \log(n))$ Laufzeit nur für Sortieralgorithmen gilt, die auf Vergleichen basieren. Beispielsweise lässt sich mit Radix-Sort eine lineare Laufzeit erzielen (wenn die Größe der Hausnummern nach oben begrenzt ist).

4 Umsetzung

Die Implementationen des ersten Ansatzes und dessen Optimierung sind ähnlich aufgebaut. Die Lösung in Linearzeit und die Erweiterung unterscheiden sich jedoch in der Struktur von den ersten beiden. Daher werden die Umsetzungen aller Lösungen behandelt. Gleich sind bei beiden aber ein paar globale Variablen: `n` und `u` geben wie in der Definition die Anzahl der Häuser und den Umfang des Sees an. In `vector<int> house` stehen die Positionen der Häuser in aufsteigend sortierter Reihenfolge. Die ersten beiden Lösungen enthalten zusätzlich noch zwei weitere Int-Vektoren:

- `votes` gibt für jede Position am See die Anzahl der Stimmen an, die dort verfügbar sind. Also `votes[i] = 1`, falls sich bei i ein Haus befindet, und sonst 0. (Größe u).
- `vpref` enthält die Präfixsumme von `votes` (vpref = votes prefix). Es gilt:

$$\sum_{i=0}^{k} votes[i] = vpref[k]$$

Diese Präfixumme wird mit der in 1.2.1 hergeleiteten Formel

$$vpref[i+1] = votes[i+1] + vpref[i]$$

vorberechnet.

4.1 Arithmetik

Da der See ein Kreis ist, wird immer Modulo u gerechnet. Bei der Subtraktion ist darauf zu achten, dass in C++ `(a - b) % u` immer $a-b$ zurück gibt, falls $a-b<0$. Es muss also korrekterweise `(a - b + u) % u` gerechnet werden.
Außerdem muss Division mit Ab- bzw. Aufrunden verwendet werden. Also $\left\lfloor \frac{a}{b} \right\rfloor$ und $\left\lceil \frac{a}{b} \right\rceil$. Für die Variante mit Abrunden kann einfach `a/b` verwendet werden, da die Division in C++ automatisch abrundet. Für die Variante mit Aufrunden wird dieser Zusammenhang verwendet:[5]

$$\left\lceil \frac{a}{b} \right\rceil = \left\lfloor \frac{a \ + \ b-1}{b} \right\rfloor$$

Für Division mit Aufrunden kann also `(a+b-1)/b` verwendet werden. Dies wird vor allem bei Division durch 2 benötigt: `(a+1)/2`.
Auch bei der Erweiterung ist der See ein Kreis. Allerdings unterstützen Fließkommazahlen in C++ keinen Modulo-Operator. Damit die Werte also immer zwischen 0 und u liegen bzw. nach u wieder bei der 0 gestartet wird, wird die Funktion `mod` verwendet, welche diese Operation für positive x so umsetzt:

$$x \mod m = x - m \left\lfloor \frac{x}{m} \right\rfloor \tag{44}$$

Für negative x wird die Funktion erneut mit $-x$ als Parameter aufgerufen.

4.2 Vorberechnung

mx In den ersten beiden Lösungen wird die Funktion `precalcMaxIntervals` aufgerufen, die die Werte von `maxInterval` vorberechnet. Dabei enthält `maxInterval[a][b]` den Wert von mx zwischen Eisbuden bei den Positionen a und b.
Für diese Vorberechnung wird zunächst mit `len = m / 2 - 1` die Größe g der zu betrachtenden Intervalle bestimmt. In der einfachen Implementation werden dann alle Subintervalle der Größe `len` getestet und am Ende das Maximum zurückgegeben.
Bei der optimierten Lösung werden mit dem Monotonie-Trick alle `maxInterval[a][b]` Werte mit einem bestimmten Abstand von a und b auf einmal berechnet. Zur Implementation des Monotonie-Tricks wird eine `deque` (Double-Ended-Queue) als Datenstruktur für die Liste verwendet, da diese in $\mathcal{O}(1)$ sowohl am Anfang als auch am Ende Elemente löschen bzw. einfügen kann. Um zu bestimmen, ob ein Wert aus der linken Fenstergrenze fällt, speichern die Elemente zusätzlich noch ihren Index. Konkret wird ein `deque<pair<int, int>>` verwaltet, wobei der zweite Wert dem Index entspricht.

[5] Ist a ein Vielfaches von b, ändert die Addition von $b-1$ nichts. Ansonsten ist $a \ + \ b-1$ größer gleich dem nächsten Vielfachen von b, weshalb aufgerundet wird.

size Bei der Lösung in Linearzeit müssen die Werte von *size* vorberechnet werden. Dies geschieht in der Funktion `precalcMaxSize` und die Werte werden in `vector<vector<int>> maxSize[2]` gespeichert. Nun soll erklärt werden, welchen *size* Wert `maxSize[s][i][d]` speichert:

- In d ist die Anzahl *sum* der betrachteten Häuser kodiert. Es gilt $sum = \lfloor \frac{n}{3} \rfloor + d - 5$. Auf diese Weise lassen sich mit $d \in [0;6]$ alle nach 1.4.3 relevanten Anzahlen von Häusern zwischen zwei Eisbuden betrachten.
- Es werden die Häuser mit Indizes $\in [i+1,\ i+size]$ betrachtet.
- Zusammen mit s und *sum* lässt sich $mx' = \lceil \frac{sum}{2} \rceil + s$ berechnen. Somit entspricht s dem Δ Wert aus 2.5.

Dieses Format ist notwendig, da bei einem Zugriff über `maxSize[mx'][i][sum]` für sämtliche kleine Werte von *sum* ungenutzter Speicherplatz benötigt würde. Dann wäre die Laufzeit bereits durch die Arraygröße $\mathcal{O}(n^3)$. Durch die kompakte Speicherung ist es aber nur $\mathcal{O}(n)$.
Die Vorberechnung passiert wie in 2.3 beschrieben. Auch hier wird für die Liste beim Monotonie-Trick eine `deque` verwendet.

4.3 Stabilität testen

Um zu testen, ob eine Verteilung von Eisbuden stabil ist, werden zwei Arrays `s` und `m` der Größe drei erstellt. Dabei enthält `s` die *sum* Werte und `m` die mx Werte der drei Intervalle zwischen je zwei Eisbuden. Die Variable `best` enthält die maximale Stimmenanzahl. Entweder ist `best = m[0] + m[1] + m[2]`, wenn in jedem Intervall genau eine Eisbude platziert wird, oder `best = m[i] + s[j]`, wenn in einem Intervall eine Eisbude platziert wird und in einem anderen zwei. In diesem Fall ist zusätzlich zu überprüfen, ob `u != j`. Falls am Ende `2*best > n`, wurde eine Mehrheit gefunden und die Verteilung ist instabil, sonst ist sie stabil.
In den ersten beiden Lösungen wird dieses Finden der maximalen Stimmenzahl in der Funktion `maxVotes` implementiert. Die Werte für `s` werden dabei über die Präfixsumme berechnet, die für `m` stammen aus den vorberechneten `maxSize` Werten.

4.4 Testen der Kombinationen

In der ersten Lösung werden einfach alle Mengen von Eisbudenpositionen `i, j, k` getestet, wobei `i < j < k`. Dies wurde über drei `for`-Schleifen realisiert, bei denen für `j` der Startwert `i+1` ist und für `k` ist er `j+1`.

Optimierung Bei der optimierten Version wird zunächst eine beliebige erste Eisbude `i` platziert. Danach wird über die mögliche Anzahl an Häusern zwischen dieser und der nächsten Eisbude iteriert. Die nächste Eisbude `j` wird dann mit dem entsprechenden Häuserabstand direkt bei einem Haus platziert. Eisbude `k` befindet sich dann bei einer Position zwischen `j` und `i`. `k` startet bei `j+1` und wird so lange modulo `u` nach rechts bewegt, bis `i` erreicht wird: `for(int k = (j+1)%u; k != i; k = (k+1)%u)`

Linearzeit Zunächst wird eine Eisbude beim `i`-ten Haus platziert. Danach wird die Anzahl an Häusern zwischen `i` und der nächsten Eisbude durchgegangen: `x` und danach die Häuser zwischen der zweiten und dritten Eisbude `y`. Im nächsten Schritt werden die Eisbuden zwei und drei jeweils bei einem Haus oder zwischen zwei platziert. Die Variable `h2` gibt an, ob sich Eisbude 2 bei einem Haus befindet (`h2=1`) oder vor einem (`h2=0`). Analog hat `h3` dieselbe Bedeutung für Haus 3. Damit werden dann die Häuser `j, k` bestimmt, bei oder vor denen sich die zweite und dritte Eisbude befinden. Dabei muss berücksichtigt werden, ob sich bei der vorherigen Eisbude ein Haus befindet, da das die Anzahl der Häuser zwischen den Eisbuden beeinflusst. Mit den Werten `i, j, k` wird dann noch die Anzahl der Häuser `z` zwischen der dritten zur ersten Eisbude berechnet.
Nach dem die ungefähren Positionen der Eisbuden bekannt sind, wird über die Werte von mx' bzw. Δ iteriert. Diese heißen für die einzelnen Intervalle `p1, p2, p3`. [6] Mit diesen Werten wird dann bereits überprüft, ob die Verteilung stabil ist. Falls ja, werden die maximalen Abstände zwischen den Eisbuden `a, b, c` aus `maxSize` geladen, mit denen dann die genauen Positionen `e1, e2, e3` der Eisbuden bestimmt werden. Dazu wird der Greedy-Ansatz verwendet, den Abstand so groß wie möglich zu wählen, sodass die Eisbuden aber dennoch in den Zwischenräumen liegen, die ihnen durch die vorherigen Werte zugeschrieben wurden.
Nach der Platzierung der Eisbuden wird erneut überprüft, ob sie die Abstände einhalten und zwischen den richtigen Häusern liegen. Für letzteres wird der Abstand der tatsächliche Eisbuden mit dem der ersten Eisbude zum Haus vor der zweiten Eisbude verglichen. So kann verhindert werden, dass einer der Abstände zu klein war und

[6] Ich bin mir aber unsicher, wieso ich die Variablen so genannt habe... Vielleicht sollte das `p` für *Plus* stehen, da zusätzlich erlaubte Stimmen getestet werden oder vielleicht für *Personen*, da es um die Anzahl der Personen geht, die dafür stimmen werden? Jedenfalls heißt die Variable aus irgendeinem Grund so.

eine Eisbude noch ein Haus weiter vor dem liegt, wo sie eigentlich sein sollte.
Falls diese Überprüfungen bestanden wurden, wurde eine stabile Verteilung gefunden.

Erweiterung Die Umsetzung ist sehr ähnlich zu dem vorherigen Abschnitt. Beim genauen Platzieren der Eisbuden wurde davor eine Eisbude in einem Zwischenraum auch tatsächlich in einem solchen platziert. Falls also *size* Größer war, als der Abstand zum nächsten Haus, wurde die Eisbude vor dem Haus platziert. Dies führt zu keinem Nachteil und wurde gemacht, um die Idee des Ansatzes besser zu veranschaulichen, dass die Eisbude entweder bei einem Haus oder in einem Zwischenraum liegt. Hier kann aber nicht 1 von der entsprechenden Hausposition abgezogen werden, da die Positionen nicht ganzzahlig sein müssen. Somit müsste bei der Implementation eigentlich ε abgezogen werden. Da dies aber die einzige Stelle ist, an der ε in der Implementation benötigt wird und das Arbeiten mit beliebig kleinen Werten leicht zu Problemen wie Rundungsfehlern führen kann, wird ein Ansatz ohne ε verwendet: Wenn eine Eisbude eigentlich vor einem Haus platziert werden soll, *size* aber theoretisch groß genug ist, wird die Eisbude genau bei dem Haus platziert und nicht davor. Dabei bleibt die Verteilung immer noch stabil, da in keinem Zwischenraum neue Häuser dazu kommen, sondern es sogar weniger werden. Somit kann sich die Anzahl der Stimmen dadurch nicht erhöhen und es lässt sich die Definition einer sehr kleinen Konstante vermeiden.

5 Beispiele

5.1 BWINF-Beispiele

Zunächst werden die Ausgaben vom lineare Lösungsansatz dargestellt. Diese unterscheiden sich möglicherweise von den Ausgaben der anderen Programme, da es mehrere stabile Verteilungen geben kann und sich die Suchreihenfolge unterscheidet.

```
eisbuden1.txt:

Stabile Verteilung gefunden!
2 8 14


eisbuden2.txt:

Es wurde keine stabile Verteilung gefunden!


eisbuden3.txt:

Stabile Verteilung gefunden!
1 17 42


eisbuden4.txt:

Es wurde keine stabile Verteilung gefunden!


eisbduden5.txt:

Stabile Verteilung gefunden!
83 130 233


eisbuden6.txt:

Es wurde keine stabile Verteilung gefunden!


eisbuden7.txt:

Stabile Verteilung gefunden!
114 285 416
```

Hier auch nochmal übersichtlicher in einer Tabelle dargestellt:

Datei	stabile Verteilung
eisbuden1.txt	2, 8, 14
eisbuden2.txt	-
eisbuden3.txt	1, 17, 42
eisbuden4.txt	-
eisbuden5.txt	83, 130, 233
eisbuden6.txt	-
eisbuden7.txt	114, 285, 416

5.2 Eigene Beispiele

Die Beispiele der BWINF-Webseiten sind bereits mit den ersten Ansätzen lösbar. Es soll aber auch die Geschwindigkeit des linearen Ansatzes demonstriert werden. Zunächst ein Beispiel, dass die Unabhängigkeit vom Umfang zeigt. Es ist in der Datei `BigU` zu finden:

```
1000000000 20
2003340 12772325 19338033 61666739 76777802 109435193 129839138 167940523 197345470
    431572218 436559835 455957972 509648610 623955781 694264499 725248279 832378816
    929054734 972297644 979491059
```

Dieses Beispiel zeichnet sich durch ein $u = 10^9$ aus mit geringer Hausanzahl. Die ersten Ansätze hätten das Beispiel nicht in unter einem Jahr gelöst[7]. Der lineare Ansatz braucht aber nur 0.004 Sekunden, da es nur 20 Häuser sind. Er kommt auf dieses Ergebnis:

```
Stabile Verteilung gefunden!
2003340 109435193 509648610
```

Zusätzlich soll noch ein Beispiel zur Erweiterung gezeigt werden:

```
10.5 10
1.2305581338524234 3.600647485100913 3.875677273398641 5.25033277534192 5.759562099083442
     5.952664740290466 7.43193558493988 7.5951339268835465 9.264072128495522
    9.388541760343069
```

Die Erweiterung kommt auf das folgende Ergebnis:

```
Stabile Verteilung gefunden!
3.600647485100913 5.759562099083442 7.431935584939880
```

Natürlich lässt sich die Erweiterung auch auf die BWINF-Beispiele anwenden. Die Ergebnisse sind erneut als Tabelle dargestellt:

Datei	stabile Verteilung
eisbuden1.txt	2.000000000000000, 8.000000000000000, 14.000000000000000
eisbuden2.txt	-
eisbuden3.txt	-
eisbuden4.txt	-
eisbuden5.txt	-
eisbuden6.txt	-
eisbuden7.txt	114.000000000000000 285.000000000000000 416.000000000000000

Interessant ist dabei, dass für die Beispiele 1 und 7 identische Lösungen gefunden wurden, wie zuvor, für Beispiele 3 und 5 nun aber keine stabile Verteilung mehr existiert. Letzteres lässt sich intuitiv damit erklären, dass es mit reellen Eisbudenpositionen deutlich mehr mögliche Vorschläge gibt, die angenommen werden könnten. Etwas formaler lässt sich das mit dem größeren Wert von g erklären. Bei gleichem Abstand von zwei Eisbuden können in einem größeren Bereich stimmen geholt werden, indem eine Eisbude zwischen zwei Häusern platziert wird:

		E	H	H	E														

Wird nun Eisbude zwischen den beiden Häusern platziert, verringert sich ihr Abstand auf 0,5 Einheiten. Darf aber keine Eisbude dazwischen platziert werden, kann sie nur bei einem Haus stehen. Für das andere Haus würde sich dann am Abstand 1 nichts ändern.

[7] Geht man von 10^8 Operationen pro Sekunde aus, sind es bei $\mathcal{O}(u^2)$ etwa 10^10 Sekunden. Das sind 115740 Tage, also etwa 317 Jahre.

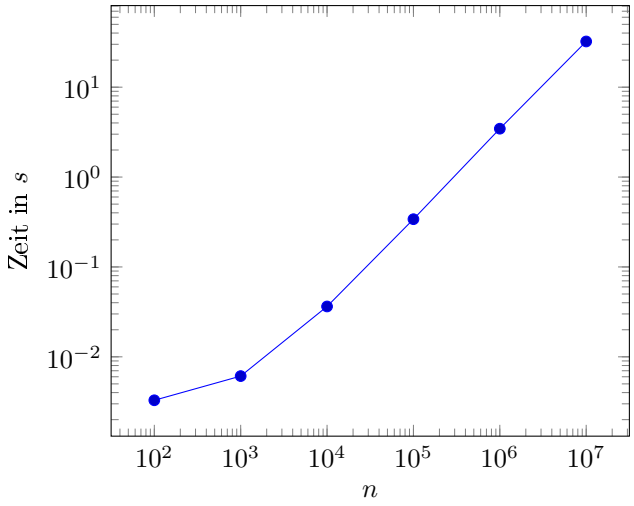


Abbildung 3: Laufzeit in Abhängigkeit von n. Die Achsen sind logarithmisch skaliert.

5.3 Empirische Betrachtung

Nun wird für verschiedene Anzahlen der Häuser jeweils 50 mal gemessen, wie lange das Programm benötigt und der Durchschnitt gebildet, um die Laufzeit in Abhängigkeit von n zu untersuchen. Gleichzeitig wird überprüft, wie oft es eine stabile Verteilung gab. Hier sind die Ergebnisse meines Tests[8]:

```
n=100
Stabile Verteilungen: 1
Durchschnittliche Zeit:  0.003285846710205078
n=1000
Stabile Verteilungen: 0
Durchschnittliche Zeit:  0.00610506534576416
n=10000
Stabile Verteilungen: 0
Durchschnittliche Zeit:  0.03633328437805176
n=100000
Stabile Verteilungen: 0
Durchschnittliche Zeit:  0.3398170304298401
n=1000000
Stabile Verteilungen: 0
Durchschnittliche Zeit:  3.4520682501792908
```

Für $n = 10^7$ wurde nur ein Test durchgeführt, da es sonst zu lange gedauert hätte. Das Ergebnis war eine Zeit von $32.252s$. In Abbildung 3 sind die Werte auch noch einmal als Graph dargestellt. Die Daten bestätigen die Laufzeit von $\mathcal{O}(n)$, da ein Verzehnfachen von n etwa einem Verzehnfachen von der Zeit entspricht: $0.339 \cdot 10 \approx 3.452$. Neben der linearen Laufzeit lässt sich aber auch ein hoher konstanter Faktor beobachten. Denn bereits bei $n = 10^6$ werden etwa 3 Sekunden benötigt. Bei einem recht kleinen / gewöhnlichen konstanten Faktor wird im Competitive Programming häufig die Faustregel von 10^8 Operationen pro Sekunde genutzt. Hat ein Programm also eine Laufzeit von $\mathcal{O}(n^2)$, wird angenommen, dass es bei $n < 10000$ in unter einer Sekunde fertig ist. Bei der Laufzeit von $\mathcal{O}(n)$ wäre die Annahme bei $n = 10^6$ 0.01 Sekunden. Tatsächlich waren es aber 3.452, also mehr als das 300fache. Dieser hohe konstante Faktor lässt sich damit erklären, dass die Lösung zum erreichen der linearen Laufzeit mehrfach konstant viele Möglichkeiten durchprobiert. Etwa die 7 möglichen Anzahlen von Häusern zwischen zwei Eisbuden oder die mx' bzw. Δ Werte.

Abgesehen von der Laufzeit, zeigen die Daten aber auch, dass stabile Verteilungen mit steigender Häuseranzahl n immer unwahrscheinlicher / seltener werden. Bereits bei $n = 100$ war unter 50 Verteilungen nur eine. Eine mögliche Erklärung hierfür ist, dass immer mindestens die Hälfte der Häuser, die sich aktuell nicht bei einer Eisbude befinden, für einen Vorschlag gewonnen werden können. Sobald auch nur in zwei Intervallen eine Stimme mehr geholt werden kann, führt dies direkt zu einer Mehrheit (siehe Δ-Werte aus 2.5). Und bei sehr vielen Häusern ist es wahrscheinlich, dass es Stellen mit sehr vielen Häusern und geringem Abstand gibt. Dann ist es wiederum möglich viele Stimmen auf einmal zu holen, was eine stabile Verteilung unmöglich macht. Damit es eine stabile Verteilung gibt, müssten die Häuser gleichmäßiger verteilt sein. Das ist aber nicht das, was bei einer zufälligen Auswahl passiert.

[8] Falls Sie eigene Tests durchführen wollen, können sie das Skript `tester.py` verwenden.

6 Quellcode

6.1 Allgemein

Variablen und Hilfsfunktionen:

```
int u, n; // Umfang des Sees, Anzahl der Häuser
vector<int> house; // Liste der Häuser
vector<int> votes; // Stimmenanzahl bei Index; 1, wenn sich dort ein Haus befindet, sonst 0
vector<int> vpref; // Präfixsumme von votes

// Bestimmt in O(1) die Anzahl an Häusern / Stimmen Zwischen den Häusern bei l und r
// Die Grenzen sind einschließlich
int getVotes(int l, int r)
{
  if(r < l) return getVotes(0, r) + getVotes(l, u-1);
  if(l > 0) return vpref[r] - vpref[l-1];
  return vpref[r];
}

// maxInterval[i][j]:
// Maximale Anzahl an Stimmen, die bei einem Vorschlag mit einer Eisbude zwischen i und j
// geholt werden kann, wenn sich bei i und j eine Eisbude befindet.
vector<vector<int>> maxInterval;

// Addition modulo u
int add(int a, int b)
{
  return (a + b) % u;
}

// Subtraktion modulo u
int sub(int a, int b)
{
  return (a - b + u) % u;
}
```

Finden der maximalen Stimmenanzahl:

```
// Eisbuden bei Positionen i, j, k
// Berechnet die maximale Stimmenanzahl, die ein Vorschlag erhalten kann
int maxVotes(int i, int j, int k)
{
  // Anzahl an Stimmen, die mit 2 Eisbuden geholt werden können
  // (= Anzahl an Häusern zwischen den Eisbuden)
  int s[3];
  s[0] = getVotes(add(i, 1), sub(j, 1));
  s[1] = getVotes(add(j, 1), sub(k, 1));
  s[2] = getVotes(add(k, 1), sub(i, 1));
  // Anzahl an Stimmen, die mit einer Eisbude geholt werden können
  int m[3];
  m[0] = maxInterval[i][j];
  m[1] = maxInterval[j][k];
  m[2] = maxInterval[k][i];
  // Teste alle Möglichkeiten, die 3 Eisbuden auf die Intervalle zu verteilen
  // (geht nur, weil es so wenige sind!)
  int best = m[0] + m[1] + m[2]; // Je eine Eisbude pro Intervall
  for(int g = 0; g < 3; ++g)
  {
    for(int h = 0; h < 3; ++h)
    {
      // Zwei Eisbuden in einem Intervall und eine in einem anderen
      if(g != h) best = max(best, s[g] + m[h]);
    }
  }
  return best;
}
```

Vorberechnung der Präfixsumme:

```
vpref[0] = votes[0];
```

```
for(int i = 1; i < u; ++i)
{
  vpref[i] = vpref[i-1] + votes[i];
}
```

6.2 Einfache Implementation

Vorberechnung von `maxIntervals`:

```
void precalcMaxIntervals()
{
  maxInterval.assign(u, vector<int>(u));
  // Iteriere über alle Paare von Eisbudenpositionen
  for(int i = 0; i < u; ++i)
  {
    for(int j = 0; j < u; ++j)
    {
      int up = j;
      if(j < i) up += u;
      int m = up - i; // Distanz von i nach j
      // g aus der Dokumentation:
      // Größe des Intervalls, in dem die Bewohner für den Vorschlag stimmen
      int len = m / 2 - 1;
      int ans = 0;
      if(len >= 0)
      {
        // Finde von allen Intervallsummen die Größte
        for(int k = i+1; k + len < up; k++)
        {
          ans = max(ans, getVotes(k % u, (k+len) % u));
        }
      }
      maxInterval[i][j] = ans;
    }
  }
}
```

Ausprobieren alle Verteilungen:

```
// Iteriere über alle Eisbudenverteilungen
for(int i = 0; i < u; ++i)
{
  for(int j = i+1; j < u; ++j)
  {
    for(int k = j+1; k < u; ++k)
    {
      // Berechne die maxiamle Anzahl an Stimmen eines Vorschlages
      int best = maxVotes(i, j, k);
      // Stabile Verteilung gefunden, falls keine Mehrheit erreicht werden kann
      if(best * 2 <= n)
      {
        cout << "Stabile Verteilung gefunden!\n";
        cout << i << " " << j << " " << k << "\n";
        exit(0);
      }
    }
  }
}

cout << "Es wurde keine stabile Verteilung gefunden!\n";
```

6.3 Optimierte Implementation

Vorberechnung von `maxIntervals`:

```
void precalcMaxIntervals()
{
```

```
  maxInterval.assign(u, vector<int>(u));
  // m: Abstand der beiden Eisbuden / Größe des Intervalls
  // m < 2 nicht relevant, da sich dann keine Eisbude dazwischen befinden kann
  for(int m = 2; m < u-1; ++m)
  {
    // g aus der Dokumentation:
    // Größe des Intervalls, in dem die Bewohner für den Vorschlag stimmen
    int len = m / 2 - 1;
    // deque für den Monotonie-Trick
    deque<pair<int, int>> q;
    // Initialisieren der deque
    for(int k = 1; k + len < m; k++)
    {
      // Stimmenanzahl in einem Intervall der Größe g bzw. len
      int val = getVotes(k % u, (k+len) % u);
      // Entferne das letzte Element, solange es kleiner ist als das aktuelle
      while(!q.empty() && q.back().first < val) q.pop_back();
      q.emplace_back(val, k);
    }
    // Vorne in der deque steht die maximale Intervallsumme mx
    maxInterval[0][m] = q.front().first;

    // Gehe alle Positionen für die linke Eisbude durch
    for(int i = 1; i < u; ++i)
    {
      // Summe des Intervalls, das direkt vor der rechten Eisbude bei i+m endet
      int val = getVotes(sub(i + m - 1, len), add(i, m-1));
      // Entferne das vordere Element, falls es aus links aus dem Fenster fällt
      if(q.front().second <= i) q.pop_front();
      while(!q.empty() && q.back().first < val) q.pop_back();
      q.emplace_back(val, i+m-len-1);

      maxInterval[i][add(i, m)] = q.front().first;
    }
  }
}
```

Ausprobieren der relevanten Verteilungen:

```
  // Position der 1. Eisbude
for(int i = 0; i < u; ++i)
{
  // Anzahl an Häusern zwischen i und j
  for(int hdist = -5; hdist <= 1; ++hdist)
  {
    // Haus, das nach i kommt. Der Index entspricht der Anzahl an Häsuern vor oder bei i
    int houseAfter = vpref[i];
    // Position der 2. Eisbude
    // Diese befindet sich bei einem Haus -> Berechnung über Index des Hauses
    int j = house[(houseAfter + n/3 + hdist + n) % n];

    // Position der 3. Eisbude (nach j und vor i)
    for(int k = (j+1)%u; k != i; k = (k+1)%u)
    {
      // Berechne die maxiamle Anzahl an Stimmen eines Vorschlages
      int best = maxVotes(i, j, k);
      // Stabile Verteilung gefunden, falls keine Mehrheit erreicht werden kann
      if(best * 2 <= n)
      {
        // Es wurde eine Lösung gefunden!
        // Sortiere die Positionen und gib sie aus!
        vector<int> ans = {i, j, k};
        sort(ans.begin(), ans.end());
        cout << "Stabile Verteilung gefunden!\n";
        cout << ans[0] << " " << ans[1] << " " << ans[2] << "\n";
        exit(0);
      }
    }
  }
}

```

```
cout << "Es wurde keine stabile Verteilung gefunden!\n";
```

6.4 Lineare Lösung

Vorberechnung der *size* Werte:

```
// maxSize[s][i][j]:
// Maximaler Abstand der Eisbude bei Haus i von einer nach Haus i+n/3-5+1
// sodass genau s Stimmen mehr geholt werden als die Hälfte (ceil) der Anzahl der Häuser zwischen den E
vector<vector<int>> maxSize[2];

// Vorberechnung von maxSize
void precalcMaxSize()
{
  // Iteriere über die mx' bzw. Delta-Werte
  for(int s = 0; s < 2; ++s)
  {
    maxSize[s].assign(n, vector<int>(7));

    // Iteriere über die relevanten Anzahlen an Häusern zwischen zwei Eisbuden
    for(int d = 0; d < 7; ++d)
    {
      // Anzahl an Häusern zwischen zwei Eisbuden
      int sum = n/3 + (d-5);
      if(sum < 0) continue; // Die Anzahl an Häusern kann nicht negativ sein
      // mx' aus der Doku
      // Stimmenzahl mit einer Eisbude, die nicht überschritten werden darf
      int wanted = (sum + 1) / 2 + s;
      // Falls die Anzahl der Häuser nicht größer als mx' ist,
      // wird mx' sicher nie überschritten -> size beliebig groß
      if(wanted >= sum)
      {
        for(int i = 0; i < n; ++i) maxSize[s][i][d] = u;
        continue;
      }

      // deque für Monotonie-Trick
      deque<pair<int, int>> q;
      // Index vom rechten Haus. Es werden alle Häuser in (i, j] betrachtet
      int j = sum;
      // Initialisieren der deque
      for(int i = 1; i + wanted <= j; ++i)
      {
        // Abstand der Häuser mit Indexdifferenz mx'
        int val = (house[(i+wanted) % n] - house[i] + u) % u;
        // Entferne das letzte Element der deque, solange es nicht kleiner val ist
        while(!q.empty() && val <= q.back().first) q.pop_back();
        q.emplace_back(val, i);
      }
      // G steht vorne in der deque
      // Es gilt: size = 2 * G + 1
      maxSize[s][0][d] = 2*q.front().first+1;

      for(int i = 1; i < n; ++i)
      {
        j = i + sum;
        int val = (house[j%n] - house[(j-wanted+n)%n] + u) % u;
        // Entferne das vordere Element, falls es aus links aus dem Fenster fällt
        if(q.front().second <= i) q.pop_front();
        while(!q.empty() && val <= q.back().first) q.pop_back();
        q.emplace_back(val, j-wanted);

        maxSize[s][i][d] = 2*q.front().first+1;
      }
    }
  }
}
```

Ausprobieren der Verteilungen:

```
// Erste Eisbude
for(int i = 0; i < n; ++i)
{
  // Anzahl an Häusern zwischen der ersten und zweiten Eidbude (+ n/3)
  for(int x = -min(5, n/3); x <= 1; ++x)
  {
    // Anzahl an Häusern zwischen der zweiten und dritten Eidbude (+ n/3)
    for(int y = -min(5, n/3); y <= 1; ++y)
    {
      // Befindet sich Eisbude 2 bei einem Haus (h2 = 1)
      // Oder VOR dem angegebenen Haus (h2 = 0)
      for(int h2 = 0; h2 < 2; ++h2)
      {
        // Wie h2, nur für Eisbude 3
        for(int h3 = 0; h3 < 2; ++h3)
        {
          // k soll i nicht "überrunden"
          if(2 * n/3 + x + y + 1 + h2 >= n) continue;

          // Bestimmen der Eisbudenpositionen j, k

          // Es befinden sich n/3 + x Häuser zwischen i und j
          // i steht bei einem Haus -> 1 addieren
          int j = (i + n/3 + x + 1) % n;
          // Falls j bei einem Haus steht muss 1 addiert werden
          int k = (j + n/3 + y + h2) % n;

          // Abstand zwischen Eisbuden k und i
          // Wie davor gilt: i = (k + n/3 + z + h3)
          // Umstellen ergibt z
          int z = (i-k+n)%n - n/3 - h3;
          if(z < -5 || z > 1) continue;

          // sum Werte für Testen der Stabilität
          int s[3];
          s[0] = x + n/3;
          s[1] = y + n/3;
          s[2] = z + n/3;
          // Delta / mx' Werte
          for(int p1 = 0; p1 < 2; ++p1)
          {
            for(int p2 = 0; p2 < 2 - p1; ++p2)
            {
              for(int p3 = 0; p3 < 2 - p2 - p1; ++p3)
              {
                // mx Werte für Stabilitätstest
                int m[3];
                m[0] = (s[0] + 1) / 2 + p1;
                m[1] = (s[1] + 1) / 2 + p2;
                m[2] = (s[2] + 1) / 2 + p3;

                // maximale Abstände zwischen den Eisbuden / size Werte
                int a = maxSize[p1][i][x+5];
                int b = maxSize[p2][(j-1+h2+n)%n][y+5];
                int c = maxSize[p3][(k-1+h3+n)%n][z+5];

                // Überprüfe ob stabil
                int best = m[0] + m[1] + m[2]; // Je eine Eisbude pro Intervall
                for(int g = 0; g < 3; ++g)
                {
                  for(int h = 0; h < 3; ++h)
                  {
                    // Zwei Eisbuden in einem Intervall und eine in einem anderen
                    if(g != h) best = max(best, s[g] + m[h]);
                  }
                }
                // Nicht stabil!
                if(best * 2 > n) continue;

                // Bestimme die genauen Eisbudenpositionen e1, e2, e3
                int e1 = house[i];
                int e2, e3;
                if(h2) e2 = house[j % n];
```

```
                else e2 = (e1 + min(a, (house[j] - e1 - 1 + u) % u)) % u;
                if(h3) e3 = house[k % n];
                else e3 = (e2 + min(b, (house[k] - e2 - 1 + u) % u)) % u;

                // Die Eisbuden werden so platziert, dass sie nicht nach dem Haus kommen,
                // bei dem sie sein sollten. Nun wird überprüft, ob sie nach dem vorherigen
                // Haus kommen -> liegen sie zwischen den richtigen Häusern?
                if((house[(j-1+n)%n] - e1 + u) % u >= (e2 - e1 + u) % u) continue;
                if((house[(k-1+n)%n] - e2 + u) % u >= (e3 - e2 + u) % u) continue;

                // Finale Überprüfung, ob die Abstände der Eisbuden klein genug sind
                if((e2 - e1 + u) % u > a) continue;
                if((e3 - e2 + u) % u > b) continue;
                if((e1 - e3 + u) % u > c) continue;

                // Es wurde eine Lösung gefunden!
                // Sortiere die Positionen und gib sie aus!
                vector<int> ans = {e1, e2, e3};
                sort(ans.begin(), ans.end());
                cout << "Stabile Verteilung gefunden!\n";
                cout << ans[0] << " " << ans[1] << " " << ans[2] << "\n";
                exit(0);
              }
            }
          }

        }
      }
    }
  }
}
cout << "Es wurde keine stabile Verteilung gefunden!\n";
```

6.5 Erweiterung

Die Implementation der Erweiterung unterscheidet sich nur an sehr wenigen Stellen von der linearen. Der häufigste Unterschied ist die Verwendung von `double` statt `int`. Deshalb ist hier nicht der gesamte Quellcode abgedruckt:

```
// Modulorechnung mit double Werten
double mod(double x, double m)
{
  if(x < 0) return m - mod(-x, m);
  else return x - (m * floor(x / m));
}

...

// In precalcMaxSize

  // G steht vorne in der deque
  // Es gilt: size = 2 * G
  maxSize[s][0][d] = 2*q.front().first;

...

// Bestimme die genauen Eisbudenpositionen e1, e2, e3
double e1 = house[i];
double e2, e3;
if(h2) e2 = house[j % n];
else e2 = mod(e1 + min(a, mod(house[j] - e1, u)), u);
if(h3) e3 = house[k % n];
else e3 = mod(e2 + min(b, mod(house[k] - e2, u)), u);
```