MafI1 - Weihnachtsaufgabe

208225 David Schmidt, david3.schmidt@tu-dortmund.de

Gruppe: 4, Mi. 8-10 bei Simon Demes,
Abgabe: 11.01.2019, 14:00 Uhr

1 Vorüberlegungen

1.1 Vereinfachung der Ausgangsformel für $\vec{a}, \vec{b} \in M(n)$

$$A_\Delta(\vec{a},\vec{b}) = \frac{1}{2}\sqrt{(|\vec{a}|\cdot|\vec{b}|)^2 - (\vec{a}\bullet\vec{b})^2} \qquad \cdot 2, ()^2$$

$$4\cdot A_\Delta(\vec{a},\vec{b})^2 = (|\vec{a}|\cdot|\vec{b}|)^2 - (\vec{a}\bullet\vec{b})^2 \qquad |\vec{a}| = \sqrt{a_1^2+a_2^2+...+a_n^2}$$

$$= (\sqrt{a_1^2+...+a_n^2}\cdot\sqrt{b_1^2+...+b_n^2})^2 - (\vec{a}\bullet\vec{b})^2 \qquad \forall a_i \in \{0,1\}.\, a_i^2 = a_i$$

$$= (\sqrt{a_1+...+a_n})^2\cdot(\sqrt{b_1+...+b_n})^2 - (\vec{a}\bullet\vec{b})^2$$

$$= \sum_{i=1}^{n} a_i \cdot \sum_{i=1}^{n} b_i - (\vec{a}\bullet\vec{b})^2 \qquad \vec{a}\bullet\vec{b} = \sum_{i=1}^{n}(a_i\cdot b_i)$$

$$= \sum_{i=1}^{n} a_i \cdot \sum_{i=1}^{n} b_i - (\sum_{i=1}^{n}(a_i\cdot b_i))^2$$

1.2 Abschätzung der Ausgangsformel für $\vec{a}, \vec{b} \in M(n)$

Da $\vec{a},\vec{b} \in \{0,1\}^n$ ist $4A_\Delta(\vec{a},\vec{b})^2 \in \mathbb{N}$. Laut Aufgabenstellung soll gelten:

$$A_\Delta(\vec{a},\vec{b}) < 19$$
$$\iff 4(A_\Delta(\vec{a},\vec{b}))^2 < 1444$$
$$\iff \sum_{i=1}^{n} a_i \cdot \sum_{i=1}^{n} b_i - (\sum_{i=1}^{n}(a_i\cdot b_i))^2 < 1444$$

Da wie oben beschrieben nur natürliche Zahlen in Frage kommen, ist die kleinste natürliche Zahl, die diese Beziehung erfüllt, offensichtlich der Vorgänger von 1444, also 1443. Somit ergibt sich für die zu minimierende Differenz von $19 - A_\Delta(\vec{a},\vec{b}) = 19 - \frac{1}{2}\sqrt{1443} \approx 19 - 18,99341991 \approx 6,580086778\cdot 10^{-3}$
Da zwischen 1443 und 1444 keine natürliche Zahl liegt, und wir trivialerweise in $4(A_\Delta(\vec{a},\vec{b}))^2$ keine negativen Werte erhalten können (die ohnehin nur zu einer

Vergrößerung der zu minimierenden Differenz führen würden), kommen wir mit $A_\Delta(\vec{a}, \vec{b})$ nicht näher an die 19 heran.

$$4(A_\Delta(\vec{a}, \vec{b}))^2 = \underbrace{\sum_{i=1}^{n} a_i \cdot \sum_{i=1}^{n} b_i}_{(1)} - \underbrace{(\sum_{i=1}^{n}(a_i \cdot b_i))^2}_{(2)}$$

Um sich diesen einfachen Umstand zu verdeutlichen, ist es sinnvoll, sich zu überlegen, dass wir in (1) lediglich die Einsen in $\vec{a}$ und $\vec{b}$ zählen und multiplizieren. In (2) zählen wir die Positionen in den Vektoren, wo in $\vec{a}$ und $\vec{b}$ eine 1 steht (im Grunde also ein logisches AND). Dieser Wert wird dann noch quadriert. Für die ganze Formel gilt also, dass wir mit den Zahlen 0 und 1 sowie den Grundrechenarten nie die ganzen Zahlen verlassen (was offensichtlich sein sollte). Die natürlichen Zahlen verlassen wir ebenfalls nicht, da Folgendes gilt:

$$\sum_{i=1}^{n}(a_i \cdot b_i) \leq min(\sum_{i=1}^{n} a_i, \sum_{i=1}^{n} b_i)$$

$$\iff (\sum_{i=1}^{n}(a_i \cdot b_i))^2 \leq (min(\sum_{i=1}^{n} a_i, \sum_{i=1}^{n} b_i))^2 \leq \sum_{i=1}^{n} a_i \cdot \sum_{i=1}^{n} b_i$$

Es können offensichtlich nicht an mehr Stellen Einsen in $\vec{a}$ UND $\vec{b}$ stehen, als in einem der beiden Vektoren überhaupt Einsen stehen.
Damit ist gezeigt, dass $4(A_\Delta(\vec{a}, \vec{b}))^2 \in \mathbb{N}$ und damit auch, dass 1443 die kleinste so erreichbare Zahl < 1444 ist, wir also nicht näher an die 19 herankommen, ohne sie zu erreichen, was genau unserer Zielsetzung entspricht.

2 Triviale Lösung für $\vec{a}, \vec{b} \in M(n)$

Eine triviale Lösung ergibt sich direkt für die Dimension $n = 1444$. Wir suchen zwei Vektoren, deren Produkt der Anzahl der Einsen gerade 1443 ergibt, deren Skalarprodukt aber 0 ist, also an keiner Stelle in beiden Vektoren eine 1 steht. Dies trifft in Dimension 1444 für folgende Vektoren gerade zu:

$$\vec{a} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ ... \end{pmatrix} \text{ und } \vec{b} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ ... \end{pmatrix}$$

Anders notiert: $a_i = \begin{cases} 1 & \text{für } i = 1 \\ 0 & \text{für } 2 \leq i \leq 1444 \end{cases}$ $b_i = \begin{cases} 0 & \text{für } i = 1 \\ 1 & \text{für } 2 \leq i \leq 1444 \end{cases}$

Genau genommen trifft es sogar für jeden Vektor zu in dem gilt: $\vec{a}, \vec{b} \in \{0, 1\}^{1444}$ mit

$$\vec{a} = \begin{pmatrix} a_1 \\ a_2 \\ ... \\ a_{1444} \end{pmatrix}, \vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ ... \\ b_{1444} \end{pmatrix} \text{ und } \sum_{i=1}^{n}(a_i) = 1 \text{ und } \sum_{i=1}^{n}(b_i) = 1443 \text{ und}$$

$\forall i \in \mathbb{N}, 1 \leq i \leq 1444. a_i \oplus b_i$ ($\oplus$ steht hier für das logische XOR, wenn man $1 = T$ und $0 = F$ interpretiert).
Damit haben wir bereits 1444 mögliche Lösungen für das Problem gefunden, jedoch ist die Dimension mit 1444 auch sehr hoch. Somit versuchen wir im Folgenden, die kleinste Dimension zu finden, mit der wir mit Bitvektoren maximal nah an die 19 herankommen.

3 Notwendiges Kriterium für die minimale Dimension

Offensichtlich muss $\sum_{i=1}^{n} a_i \cdot \sum_{i=1}^{n} b_i$ mindestens 1443 sein, damit wir die 1443 erreichen können, da Teil (2) ebenfalls eine natürliche Zahl ist (Quadrat einer Summe von Einsen) und sich so in jedem Fall "negativ" auf das Gesamtergebnis auswirkt. Die größte Zahl, die wir ohne Beachtung von (2) in (1) für jede Dimension erreichen können, ist gerade, wenn alle Einträge in beiden Vektoren auf 1 stehen. Daraus ergibt sich eine notwendige minimale Dimension von
$\lceil \sqrt{1443} \rceil \approx \lceil 37,98683983 \rceil = 38$
Denn mit $38 \cdot 38 = 1444$ sind wir gerade über 1443. Alle Vektoren mit einer Dimension, die kleiner als 38 ist, kann also schon wegen Teil (1) nicht den Wert 1443 annehmen. Dabei handelt es sich jedoch nur um ein notwendiges Kriterium, da wir noch nicht gezeigt haben, dass in diesem Fall auch zwei Vektoren existieren müssen, sodass $4(A_\Delta(\vec{a}, \vec{b}))^2 = 1443$ gilt.

4 Weiterführende Überlegungen zur minimalen Dimension

Eine kurze Überlegung zeigt uns jedoch, dass es in Dimension 38 noch nicht möglich ist, genau auf den Wert 1443 zu kommen. Um in (1) überhaupt auf bzw. über 1443 zu kommen, müssen in Dimension 38 zwangsläufig alle Einträge auf 1 stehen. Damit gilt in Teil (2) jedoch auch:
$(\sum_{i=1}^{38}(a_i \cdot b_i))^2 = (\sum_{i=1}^{38}(1))^2 = 38^2 = 1444$
Da dieser Teil jedoch von (1) abgezogen wird, ergibt sich für $4(A_\Delta(\vec{a}, \vec{b}))^2$ gerade ein Wert von 0, also offensichtlich nicht 1443. Wir müssen uns also (in Anlehnung an das Schubfachprinzip) Gedanken dazu machen, wie groß $4(A_\Delta(\vec{a}, \vec{b}))^2$ in einer bestimmten Dimension maximal sein kann.
Erst wenn dieser Maximalwert größer oder gleich 1443 ist, lohnt es sich überhaupt, nach konkreten Vektoren zu suchen.
Als Grundlage dieser Überlegungen sei wie üblich n die Anzahl der Dimensionen,

$$\vec{a} = \begin{pmatrix} a_1 \\ a_2 \\ \dots \\ a_n \end{pmatrix}, \vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{pmatrix} \text{ mit } \vec{a}, \vec{b} \in M(n)$$

$e_a, e_b \in \mathbb{N}$ mit $0 \leq e_a, e_b \leq n$ die Anzahl der Einsen in $\vec{a}$ bzw. $\vec{b}$
Dann entspricht (1) gerade $e_a \cdot e_b$. (2) variiert je nach Verteilung der Einsen auf den Vektor.

4.1 Fall $n - e_a \leq e_b$ und $n - e_b \leq e_a$

Damit $4(A_\Delta(\vec{a}, \vec{b}))^2$ maximal wird, muss entsprechend (2) bzw. $(\sum_{i=1}^{n}(a_i \cdot b_i))^2$ minimal werden. Dies ist gerade dann der Fall, wenn überall da, wo im einen Vektor eine 0 steht, im anderen eine 1 steht. Dies setzt natürlich voraus, dass $n - e_a \leq e_b$ und $n - e_b \leq e_a$, also dass es in keinem Vektor mehr Nullen gibt, als es im Anderen Einsen gibt.
Gelte nun also $n - e_a \leq e_b$ und $n - e_b \leq e_a$. Dann gilt: $\exists \vec{a}, \vec{b} \in M(n). \forall i \in \mathbb{N}, 1 \leq i \leq n. (a_i = 1 \wedge b_i = 0) \vee (a_i = 0 \wedge b_i = 1) \vee (a_i = b_i = 1)$.
Dann gilt:
$min(\sum_{i=1}^{n}(a_i \cdot b_i)) = \underbrace{n - ((n - e_a) + (n - e_b))}_{(3)} = n - (2n - e_a - e_b) = e_a + e_b - n$
Zur Erklärung von (3): Da wir dafür gesorgt haben, dass möglichst wenige Stellen übrig bleiben, an denen in beiden Vektoren eine Eins steht, können wir einfach die Anzahl der Nullen in beiden Vektoren addieren. Einsen stehen dann nur an allen anderen Stellen, also gerade an $n - \underbrace{((n - e_a) + (n - e_b))}_{\text{Gesamtanzahl Nullen}}$ Stellen.
Dies entspricht in diesem Fall dem Skalarprodukt. Grundsätzlich könnte $n - (2n - e_a - e_b)$ auch negativ werden, wenn $e_a, e_b << n$ sind. Aufgrund der Voraussetzung oben ist dies jedoch nicht möglich. Damit gilt auch für (2):
$min((\sum_{i=1}^{n}(a_i \cdot b_i))^2) = (min(\sum_{i=1}^{n}(a_i \cdot b_i)))^2 = (e_a + e_b - n)^2$
Es gilt also $max(4(A_\Delta(\vec{a}, \vec{b}))^2) = e_a \cdot e_b - min((\sum_{i=1}^{n}(a_i \cdot b_i))^2) = e_a \cdot e_b - (e_a + e_b - n)^2$

4.2 Fall $n - e_a > e_b$ oder $n - e_b > e_a$

Ist die Voraussetzung vom ersten Fall nicht gegeben, ist (2) automatisch 0, wenn man sicherstellt, dass $\nexists i \in \mathbb{N}, 1 \leq i \leq n. a_i = b_i = 1$. Dies ist offensichtlich nur im oben genannten Sonderfall möglich, da es (nach bzw. in Anlehnung an das Schubfachprinzip) sonst mindestens eine Position gibt, an der beide Vektoren eine Eins enthalten. In diesem Sonderfall ist $max(4(A_\Delta(\vec{a}, \vec{b}))^2) = e_a \cdot e_b$.

5 Computergestützte Findung der tatsächlichen minimalen Dimension und einiger Beispiele

Dass all die obigen Überlegungen notwendig waren, kann man sich leicht vor Augen führen, wenn man die riesige Anzahl an Möglichkeiten beachtet, zwei Bitvektoren mit Dimensionen zwischen 38 und 1444 zu wählen. Alleine für die

Dimension 1444 ergäben sich 2^{1444} mögliche Bitvektoren, also eine unvorstellbar große Zahl.
Wir wollen also zunächst klein anfangen und die kleinste Dimension finden, in der es eine Kombination mit $max(4(A_\Delta(\vec{a},\vec{b}))^2) \geq 1443$ gibt:

```
void findFirstMax() {
    for(int n = 1; n <= 1444; ++n) {//dimensions
        //starting with 38 would work, too
        // -> others break because >n
        for(unsigned int onesInA = 1;
                onesInA <= n;
                ++onesInA) {
            //1 if we need to round up
            //because there is a remaining
            unsigned int roundUp = 1;

            //true if there is no remaining
            if(1443 % onesInA == 0)
                roundUp = 1;

            unsigned int newStart = 1443
                                    / onesInA
                                    + roundUp;

            for(unsigned int onesInB = newStart;
                    onesInB <= n;
                    ++onesInB) {
                unsigned int scalMin = onesInA
                                       + onesInB
                                       - n;
                unsigned int max = onesInA * onesInB
                                   - scalMin * scalMin;

                if(max >= 1443) {
                    std::cout << "N: " << n
                              << " A: " << onesInA
                              << " B: " << onesInB
                              << " - " << max
                              << std::endl;
                    return;
                }
            }
        }
    }
}
```

Dieses Programm liefert uns nach kurzer Zeit folgendes Ergebnis:

```
N: 66 A: 41 B: 44 - 1443
```

Wenn in Dimension 66 einer der beiden Vektoren also 41 Einsen enthält und der andere genau 44 Einsen, dann ist der maximal für $4(A_\Delta(\vec{a},\vec{b}))^2$ erreichbare Wert genau 1443. Das heißt aber auch, dass in einer Dimension, die kleiner ist als 66, der nächstmögliche Wert an 19 nicht erreichbar ist.
Wenn wir uns an die Überlegungen zum maximalen Wert erinnern, bemerken wir schnell, dass wenn das Maximum genau der gesuchte Wert ist, an jeder Position, an der in einem Vektor eine Null steht, im Anderen eine Eins stehen muss. Dies ergibt z.B. folgenden Vektor als eine Lösung für unser Problem:
$(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,$
$1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)^t = \vec{a}$
$(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,$
$0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)^t = \vec{b}$

Anders notiert: $a_i = \begin{cases} 0 & \text{für } 1 \leq i \leq 25 \\ 1 & \text{für } 26 \leq i \leq 66 \end{cases}$ $b_i = \begin{cases} 1 & \text{für } 1 \leq i \leq 25 \\ 0 & \text{für } 26 \leq i \leq 47 \\ 1 & \text{für } 48 \leq i \leq 66 \end{cases}$

Dann gilt:

$$
\begin{aligned}
&A_\Delta(\vec{a},\vec{b}) = \frac{1}{2}\sqrt{(|\vec{a}| \cdot |\vec{b}|)^2 - (\vec{a} \bullet \vec{b})^2} \\
&= \frac{1}{2}\sqrt{(\sqrt{\sum_{i=1}^{66} a_i} \cdot \sqrt{\sum_{i=1}^{66} b_i})^2 - (\sum_{i=1}^{66}(a_i \cdot b_i))^2} \\
&= \frac{1}{2}\sqrt{(\sqrt{41} \cdot \sqrt{44})^2 - (19)^2} \\
&= \frac{1}{2}\sqrt{1804 - 361} \\
&= \frac{1}{2}\sqrt{1443} \\
&\approx 18,99341991
\end{aligned}
$$

Damit ist auch ein Beispiel für ein Bitvektor-Paar in der kleinstmöglichen Dimension gefunden, die Aufgabe also gelöst. Natürlich gibt es noch deutlich mehr Beispiele, alleine in Dimension 66 muss lediglich sichergestellt werden, dass alle Nullen im jeweils anderen Vektor auf eine Eins treffen. Das Ergebnis bleibt dann unabhängig von der "Reihenfolge" der Einträge identisch.
Die Suche mit einem "Brute-Force"-Ansatz erweist sich auch mit SIMD und höchstwahrscheinlich auch mit Hardwarebeschleunigung durch eine Grafikkarte als nahezu aussichtslos. Ein Ansatz hierzu wird nach Ende der Abgabefrist auf GitHub unter dem Konto `https://github.com/dvs23/` zu finden sein. Bei

Fragen bin ich unter david3.schmidt@tu-dortmund.de erreichbar. Dieser Ansatz verwendet Multithreading sowie mit einigen eigenen Ergänzungen die SIMD-Bitset-Bibliothek von Michael Tieying-Zhang:
`https://github.com/Michael-Tieying-Zhang/SIMD-Bitset`

6 Durchgehen aller möglichen Bitvektoren mit fester Anzahl an Einsen

Als Startpunkt bietet es sich an, alle Einsen am niedrigwertigen Ende des Bitvektors zu platzieren, also "rechts": $(0,0,0,0,1,1,1)^t$. Als Zielwert bietet sich natürlich das Gegenteil an, die Platzierung am höchstwertigen Ende: $(1,1,1,0,0,0,0)^t$. Nun gilt es, alle möglichen Kombinationen zwischen diesen beiden Vektoren durchzugehen.
Hierzu habe ich zwei Möglichkeiten in Betracht gezogen, wobei sich die Zweite im weiteren Verlauf und nach einigem Profiling als deutlich geeigneter herausgestellt hat.

6.1 "Bits Verschieben"

Eine einfache, online häufig vorgeschlagene Variante bietet sich mit folgendem Algorithmus:

Data: Bitvector, z.B. (0,0,1,1,1,0)
Result: Bitvektor, z.B. (0,1,0,0,1,1)
Finde rechteste Eins, die links von einer Null steht;
if *solche Eins ist vorhanden* **then**
  tausche sie mit der Null links von ihr;
else
  Abbruch, Zielvektor schon erreicht;
end
if *rechts der getauschten Null sind noch Einsen vorhanden* **then**
  verschiebe all diese Einsen ganz nach rechts;
end

Algorithm 1: "Bits Verschieben"

Die Implementierung ist für kurze Vektoren mit etwas um- und unverständlicher "Bitmagie" umsetzbar, was jedoch für längere Vektoren nicht mehr ohne Weiteres funktioniert, da es eigentlich keinen primitiven Datentyp gibt, der mehr als 64 Bit besitzt. Da wir aber voraussichtlich erst in Dimension 66 tatsächlich anfangen, zu suchen, sind die in einschlägigen Foren vorgeschlagenen Algorithmen eher ungeeignet. Daher hier eine eigene Version mit Schleifen:

```
void nextNum(Bitset& bnum, unsigned int ndim) {
    if(ndim > 1444 //we don't need to go higher than 1444
            || bnum.Size() < ndim)//num of bits <= dims?
        throw std::runtime_error("Dimension too high!");

    int lastZero = -1;//-1 stays if no one/zero found
    int lastOne = -1;

    //if next position is still valid
    while(lastZero + 1 < ndim
            && !bnum.Get(lastZero + 1)) //and is a zero
        ++lastZero;

    //start searching for a one right after the last zero
    lastOne = lastZero;

    //protect from consecutive ones
    while(lastOne + 1 < ndim //if next pos still valid
            && bnum.Get(lastOne + 1)) //and is a one
        ++lastOne;

    if(lastOne == -1//if there is no one next to a zero
            || lastOne + 1 >= ndim)//or one is at the end
        throw std::runtime_error("Overflow!");

    //how many ones to shift to the end
    int numOnesToShift = lastOne;

    //if there were zeros before the found one
    if(lastZero != -1)//subtract them from num ones
        numOnesToShift =  lastOne - lastZero - 1;

    for(int i = 0; i < lastOne; ++i) {
        //if: set all ones at the beginning
        //else: all zeros between the last one
        //and the beginning 1es
        if(i < numOnesToShift)
            bnum.Set(i, true);
        else
            bnum.Set(i, false);
    }

    bnum.Set(lastOne, false);//change last one
    bnum.Set(lastOne + 1, true);//with zero next to it
}
```

6.2 "Reduzierte For-Schleifen"

```
void nextNum(std::vector<int>& ones, unsigned int ndim) {
    if(ndim == 0 || ones.size() == 0)
        throw std::runtime_error("Invalid parameters!");
    else if(ones[0] >= ndim - ones.size())
        throw std::runtime_error("Maximum reached!");

    //always start in most inner "loop"
    unsigned int loopPointer = ones.size() - 1;
    //reset of all inner variables required?
    bool triggerReset = false;

    //break if outest loop has reached the maximum
    while(loopPointer > 0
            || ones[0] <= ndim - ones.size()) {
        //default operation: increase index in curr loop
        ++ones[loopPointer];

        //if limit is not reached yet
        if(ones[loopPointer]
                <= ndim - (ones.size() - loopPointer))
            break;//nothing else to do, loop still valid
        else {//go up until you find a still valid loop
            //as soon as we found one, we need to
            //set all inner loops to their start value
            triggerReset = true;

            //only decrease if not already in outest loop
            if(loopPointer > 0)
                --loopPointer;
        }
    }

    //if reset triggered and the max value has not
    //been reached set all following bits back ascending
    if(triggerReset && ones[0] <= ndim - ones.size()) {
        //invalid values should be impossible
        //due to the limits of each loop
        for(unsigned int tLoop = loopPointer;
                tLoop < ones.size() - 1;
                ++tLoop)
            ones[tLoop + 1] = ones[tLoop] + 1;
    }
}
```

Die Idee hinter Variante 6.2 sind For-Schleifen, deren Schleifenvariablen in einem Array abgespeichert werden. Jede Schleifenvariable repräsentiert eine Eins im Bitvektor. Ist also eine Laufvariable z.B. 4, so wird das 5. Bit im Vektor auf 1 gesetzt (Beginn bei 0). Jede Eins bekommt also eine eigene Schleife. Die Grenzen ergeben sich eindeutig aus der Position:

```
//ones = Anzahl Einsen
//ndim = Anzahl Dimensionen
for(int i = 0; i < ndim - ones; i++) {
    for(int j = i+1; j < ndim - (ones - 1); j++) {
        ...
            for(int m = l+1; m < ndim - 2; m++) {
                for(int n = m+1; n < ndim - 1; n++) {
                    //commands
                }
            }
    }
}
```

Wie man sieht, steht die Eins eines äußeren Loops nie vor der eines inneren Loops, sodass z.B. der äußerste Loop mit "seiner" Eins nie bis ganz an das Ende kommt, sondern nur bis zur Position, die die Anzahl der Einsen vom Ende entfernt ist. Für jede Stufe, die ein Loop weiter innen ist, kommt er eine Position näher (maximal) an das Ende heran. So werden letztendlich alle Möglichkeiten abgedeckt, wenn auch in einer anderen Reihenfolge als bei 6.1:

```
6.2:
0: 00111
1: 01011
2: 10011
3: 01101
4: 10101
5: 11001
6: 01110
7: 10110
8: 11010
9: 11100
Maximum reached!
6.1:
0: 00111
1: 01011
2: 01101
3: 01110
4: 10011
5: 10101
6: 10110
7: 11001
```

```
8: 11010
9: 11100
Overflow!
```

Grundsätzlich arbeitet Variante 6.2 damit natürlich schneller, da lediglich einige Zahlen erhöht werden müssen. Da die Variante 6.1 aber direkt auf dem Bitvektor arbeitet, werden praktisch keine unnötigen Schritte gemacht, was diese Variante erstaunlicherweise schneller macht als Variante 6.2. Mit einigen Optimierungen lässt sich jedoch sicherlich auch Version 6.2 auf ein hohes Tempo bringen. Der zeitintensive Teil hier war, dass der Bitvector jedes Mal auf 0 gesetzt und dann alle Einsen neu gesetzt werden müssen. Wenn man nur die minimale Anzahl der Änderungen durchführt, wäre 6.2 sicherlich ebenfalls sehr schnell.
Zum Vergleich, auf meinem Intel i7-4770k schaffte 6.1 mit 8 Threads:

```
6.1: ca. 55000000 Bitvektor-Varianten pro Sekunde
6.2: ca. 45000000 Bitvektor-Varianten pro Sekunde
```

Die Berechnung ist vor allem daher so performant, da das Zählen der Einsen in den Vektoren sowie die Berechnung des Skalarproduktes (welche einem bitweisen AND mit anschließendem Zählen entspricht) durch die SIMD-Erweiterung AVX2 (Advanced Vector Extensions 2) unterstützt werden, die die parallele Verarbeitung von bis zu 256 Bit ermöglicht.

7 Fazit

Trotz aller Bemühungen ist die Menge der möglichen Lösungen für reines "Brute-Force" einfach zu groß. Selbst nach 1257000000000 durchprobierten Varianten wurde noch keine Lösung gefunden, auch nicht die oben vorgestellten Lösungen. Selbst wenn man sich auf Dimension 66 und 41 Einsen in $\vec{a}$ und 44 in $\vec{b}$, so ergeben sich immer noch ca.
$\binom{66}{41} \cdot \binom{66}{44} = 110052312226739737046881931391243494 4 \approx 1,1 \cdot 10^{36}$
Möglichkeiten. Bei der oben angegebenen Geschwindigkeit wären das ca.
$6,3 \cdot 10^{20}$ Jahre. Alle Möglichkeiten in Dimension 66 oder gar bis 1444 wollen wir uns gar nicht erst überlegen...
Hier konnte also eine (theoretisch) sehr rechenaufwändige Suche mit etwas logischem Denken und etwas Computerunterstützung recht schnell lösen.
Dementsprechend wünsche ich ein gutes und friedliches Jahr 2019! [1]

[1] Aus zeitlichen Gründen konnte ich nur die wesentlichen Lösungsschritte hier ausführen und nicht alles formal beweisen, weil mir dafür entweder (im Bereich Algorithmen) noch das nötige Fachwissen oder zwischen den Feiertagen und dem 35C3 die Zeit fehlte... Ich hoffe, es war trotzdem eine interessante Lektüre ;)