Aufgabe 5: Großstadtbauern Lexikografische Lösung

Team-ID: 00001

Team-Name: Was?

Bearbeiter dieser Aufgabe:
Mathieu de Borman

28. November 2025

Inhaltsverzeichnis

1 Lösungsidee

In dieser Lösung wird zunächst greedy ein möglichst guter Anbauplan ermittelt. Anschließend wird die Greedy-Suche wiederholt gemäß bestimmter Regeln aufgerufen, bis ein optimaler Anbauplan gefunden wurde.

1.1 Substring-Matching

Die Lösung einer Probleminstanz kann allein durch ein zyklisches Wort w der Länge 12 dargestellt werden. Dabei entspricht das Zeichen w_i der Zutat, deren Anbauzyklus im i-ten Monat beginnt. Ein Gericht kann gekocht werden, wenn alle drei Zutaten des Gerichts in einem der Substrings der Länge 3 aus w vorkommen. Das Problem der Aufgabenstellung entspricht also der Suche nach einem optimalen Wort. Deshalb wird nur gezeigt, wie diese Formulierung gelöst wurde.

1.2 Geordnete Gerichte

Jedes Gericht hat bis zu $3! = 6$ verschiedene Anordnungen seiner Zutaten. Für ein Gericht mit den Zutaten 0,1,2 sind das die Tupel (0,1,2),(2,1,0),(0,2,1),(1,2,0),(1,0,2),(2,0,1). Eine spezifische Anordnung der Zutaten eines Gerichts nenne ich Essen.

1.3 Binary String

Manchmal betrachte ich nur, in welchem Monat ein Gericht gekocht werden kann (1) und in welchem nicht (0). Dabei verwende ich _ für eine beliebige Belegung einer Position. Eine solche Zeichenkette könnte für eine Lösung z.B. so aussehen: $\underset{1}{1}\underset{2}{1}\underset{3}{0}\underset{4}{1}\underset{5}{1}\underset{6}{0}\underset{7}{1}\underset{8}{0}\underset{9}{1}\underset{10}{1}\underset{11}{0}\underset{12}{0}$.

1.4 Lexikografisches Maximum

In einem Teilprozess wird ermittelt, wie die Lösung mit dem lexikografisch größten Binary-String aussieht. Die Grundidee ist, dass nacheinander für jede Wortlänge von 1 bis 12 ermittelt wird, welche Teillösungen lexikografisch maximal sind. Aus den Wörtern einer Länge kann man durch Hinzufügen einer Zutat die jeweils um eins längeren maximalen Wörter ableiten.

Dieser Prozess lässt sich optimieren, indem statt Zutaten Essen hinzugefügt werden. Beispiel: Zwei Essen, bei denen das eine Gericht drei Monate nach dem anderen gekocht wird, bestimmen den Anbauplan für sechs Monate eindeutig. Es werden fünf verschiedene Fälle unterschieden, wie ein neues Essen hinzugefügt werden kann: $\underset{1\,2\,3}{111}$, $\underset{1\,2}{11}$, $\underset{1}{1}$, $\underset{1\,2}{01}$, $\underset{1\,2\,3}{001}$. Die ganz rechte Eins gehört immer zu dem neu eingefügten Essen. Wenn weniger als drei neue Zutaten hinzugefügt werden, bedeutet das, dass die ersten 1–2 Zutaten des neuen Essens mit den letzten 1–2 Zutaten des vorherigen Essens übereinstimmen. Da die Zutaten zwischen dem neu eingefügten und dem letzten Essen dadurch automatisch eindeutig sind, muss nur noch geprüft werden, welcher der fünf Fälle eintritt.

Um eine lexikografisch maximale Anordnung zu erhalten, werden die Fälle in der Reihenfolge $\underset{1\,2\,3}{111}$, $\underset{1\,2}{11}$, $\underset{1}{1}$, $\underset{1\,2}{01}$, $\underset{1\,2\,3}{001}$ ausprobiert. Solange noch unverwendete Essen existieren, funktioniert spätestens der Fall 001, da man innerhalb von drei Monaten alle Felder neu bepflanzen kann. Dieser Vorgang wird maximal bis zum Essen des 10. Monats durchgeführt, da dann bereits alle 12 Zutaten zugewiesen sind.

1.5 Bestmöglicher Anbauplan

Ist in jedem Monat etwas kochbar, ist das insgesamt optimale Ergebnis bereits erreicht. Daher betrachte ich hier nur Fälle mit weniger als 12 gekochten Gerichten. Außerdem setze ich voraus, dass es mindestens ein Gericht gibt. In dieser Lösung wird stets nach dem lexikografisch größten optimalen Anbauplan gesucht. Daher muss jeder optimale Anbauplan die Form $\underset{1}{1}\underset{2}{\_}\underset{3}{\_}\underset{4}{\_}\underset{5}{\_}\underset{6}{\_}\underset{7}{\_}\underset{8}{\_}\underset{9}{\_}\underset{10}{\_}\underset{11}{\_}\underset{12}{0}$ haben. Wäre die letzte Position eine 1, gäbe es durch eine zyklische Verschiebung stets eine lexikografisch größere Anordnung. Vor der letzten 1 dürfen maximal zwei aufeinanderfolgende Nullen stehen. Denn lägen drei aufeinanderfolgende Nullen vor (z. B. $\underset{1}{1}\underset{2}{\_}\underset{3}{\_}\underset{4}{0}\underset{5}{0}\underset{6}{0}\underset{7}{1}\underset{8}{\_}\underset{9}{\_}\underset{10}{\_}\underset{11}{\_}\underset{12}{0}$), könnte man diese durch Verschiebung ans Ende lexikografisch vergrößern (z. B. $\underset{1}{1}\underset{2}{\_}\underset{3}{\_}\underset{4}{0}\underset{5}{0}\underset{6}{1}\underset{7}{\_}\underset{8}{\_}\underset{9}{\_}\underset{10}{0}\underset{11}{0}\underset{12}{\_}$). Außerdem kann die maximale Anzahl aufeinanderfolgender Einsen nicht größer sein als die Anzahl der Einsen am Anfang des Strings.

Diese Regeln werden angewendet, um aus der lexikografisch größtmöglichen Lösung zu berechnen, wie die lexikografisch größtmögliche bessere Lösung im Binary-String maximal aussehen kann. Anschließend wird die lexikografisch maximale Lösung berechnet, die kleiner

als der betrachtete Binary-String ist. Das wird wiederholt, bis kein besserer Anbauplan mehr möglich ist.

2 Umsetzung

Die Lösung wurde in JavaScript umgesetzt.
In der Funktion `lexMax` werden die lexikografisch größten Anbaupläne ermittelt. In `nextBin` wird ermittelt, wie groß das binäre Muster im nächsten Durchgang maximal sein darf. In `patternMax` werden die Anbaupläne berechnet, die möglichst nahe an der Vorgabe aus `nextBin` liegen.

2.1 Vergleiche

Sei z die Anzahl verschiedener Zutaten. Jeder Zutat wird eine Nummer zugewiesen, sodass geprüft werden kann, ob eine Ansammlung von drei Zutaten einem Gericht entspricht. Sei a das Tripel der drei Zutaten in geordneter Form $a_1 \leq a_2 \leq a_3$. Dann lässt sich jeder Ansammlung eindeutig der Wert $a_1 + a_2 \cdot z + a_3 \cdot z^2$ zuweisen. Dadurch muss für die Überprüfung der Existenz eines Gerichts nur geprüft werden, ob eine bestimmte Zahl in einer Hashmap vorhanden ist.

3 Komplexitätsanalyse

Wir setzen voraus, dass die Grundrechenarten in konstanter Zeit durchführbar sind und dass Zahlen konstant viel Speicher belegen. Wir betrachten die Laufzeit in Abhängigkeit von der Anzahl verschiedener Gerichte g und der Anzahl verschiedener Zutaten z. Das Einlesen der Daten hat die Laufzeitkomplexität $\mathcal{O}(z + g)$.

Im schlimmsten Fall hat eine Lösung die Struktur $\underset{1}{1}\underset{2}{1}\underset{3}{0}\underset{4}{1}\underset{5}{1}\underset{6}{0}\underset{7}{1}\underset{8}{1}\underset{9}{0}\underset{10}{1}\underset{11}{\_}\underset{12}{0}$. In einem solchen Fall setzt sich jede Lösung aus genau 7 verschiedenen Gerichten zusammen. Es werden damit höchstens g^7 verschiedene Anbaupläne ermittelt. Die gesamte Laufzeitkomplexität beträgt also $\mathcal{O}(z + g^7)$.

Da die Anzahl der Monate konstant ist, existieren nur konstant viele verschiedene Binary Strings, die geprüft werden können. Die mehrfache Wiederholung der Suche nach lexikografisch maximalen Anbauplänen ist daher für die Analyse der Laufzeitkomplexität irrelevant.

4 Greedy DP Ansatz

Ein Ansatz war es, ein Greedy DP über die verschiedenen Monate zu machen. Dabei werden gespeichert, wie viele Gerichte es bis zu diesem Monat gekocht werden, was die Menge an gekochten gerichten ist, und die ersten beiden Gerichte, d.h. `dp[Monat][Gericht]=(|gekocht|, gekocht, (1. Zutat, 2. Zutat))`. Dieser Ansatz schafft auch sich wiederholende Zutaten, z.B. dass das Gericht (a, b, c) 5 Mal vorkommt.

4.1 Umsetzung

Die Zutaten werden in Integer kodiert, die Gerichte werden entsprechend als ein einziger Integer codiert. Gegeben ein Gericht $g := (a, b, c)$, ist $c(g) := n^2a + nb + c$, wobei n die Anzahl der Zutaten ist. So ist jedes Gericht eindeutig. Dabei ist ein Gericht nicht eindeutig. Es sind 6 verschiedene Anordnungen möglich, die im Array `d` inkrementiert werden. Für den 1. Monat wird `dp[1][j] = (d[j], {recomb(j)}, (j // (n ** 2), (j % (n ** 2)) // n))` für alle vorhandenen Gerichte gesetzt. Dabei gibt `recomb()` die lexicographisch kleinste Anordnung der drei Zutaten aus.

Für die weiteren Monate werden die Transition definiert, dass `dp[i][target] = (dp[i - 1][j][0] + d[target] * (rtarget not in dp[i - 1][j][1]), dp[i - 1][j][1].copy() | {rtarget}, dp[i - 1][j][2])`, wenn es so `dp[i][target]` verbessert. Die letzten zwei Monate sind vordeterminiert, es werden die entsprechenden updates in die letzten beiden Spalten des DP eingetragen. Zum Schluss wird das Maximum genommen und ausgegeben.

Die Laufzeit beträgt $\mathcal{O}(n^4)$, da in jedem Monat n^3 Paare je n mal geupdated werden.

Das Programm hat für jede Beispieleingabe den richtigen Output produziert.

5 Werkzeuge

In dieser Arbeit wurden keine KIs, ILP-Solver oder LLMs verwendet.

6 Beispiele

Wir rufen nun das JavaScript-Programm mit den verschiedenen BwInf-Eingabedateien auf. Das Programm wird mithilfe des Browsers ausgeführt. Über den Knopf „Datei hochladen“ können Beispieleingaben ausgewählt werden. Die Ausgabe erscheint auf der Webseite. Unter „cultivation plan:“ steht dann der optimale Anbauplan. Das Programm terminiert für alle Beispieleingabedateien in weniger als 200ms auf einem gewöhnlichen PC. Alle Ergebnisse sind auch im Ordner der Programmdatei zu finden.

Eingabe	Laufzeit	angebaute Gerichte
bauern1	0.009s	4
bauern2	0.002s	6
bauern3	0.022s	8
bauern4	0.052s	9
bauern5	0.053s	9
bauern6	0.186s	10
emoji	0.005s	4
amogus	0.085s	7
bauernt1	0.03s	5
bauernt2	0.01s	12
bauernt3	0.005s	4

6.1 Standardbeispiele

bauern1

```
Runtime: 0.009s
cultivation plan:
     Kartoffel Kartoffel Kartoffel Reis      Reis      Reis      Kartoffel
Kartoffel Kartoffel Kartoffel Kartoffel Kartoffel
     Kartoffel Zwiebel   Zwiebel   Zwiebel   Kartoffel Kartoffel Kartoffel
Kartoffel Kartoffel Kartoffel Kartoffel Kartoffel
     Kartoffel Kartoffel Möhre     Möhre     Möhre     Zwiebel   Zwiebel   Zwiebel
Kartoffel Kartoffel Kartoffel Kartoffel
bin: 1         1         1         1         0         0         0         0
0         0         0         0
binSum: 4
Path:
1,1,1,1,0,0,0,0,0,0,0,0 binSum: 4 Pattern: 1,1,1,1,1,1,1,1,1,1,1,1
```

bauern2

```
Runtime: 0.002s
cultivation plan:
```

```
     Möhre       Möhre       Möhre       Paprika     Paprika     Paprika
Champignons Champignons Champignons Petersilie  Petersilie  Petersilie
     Kartoffel   Kartoffel   Kartoffel   Kartoffel   Tomate      Tomate
Tomate      Zwiebel     Zwiebel     Zwiebel     Kartoffel   Kartoffel
     Kartoffel   Kartoffel   Zwiebel     Zwiebel     Zwiebel     Reis        Reis
Reis        Tomate      Tomate      Tomate      Kartoffel
bin: 1           1           1           1           0           1           0
1           0           0           0           0
binSum: 6
Path:
1,1,1,1,0,1,0,1,0,0,0,0 binSum: 6 Pattern: 1,1,1,1,1,1,1,1,1,1,1,1
```

bauern3

```
Runtime: 0.022s
cultivation plan:
     Möhre     Möhre     Möhre     Paprika   Paprika   Paprika   Pastinake
Pastinake Pastinake Apfel     Apfel     Apfel
     Paprika   Zwiebel   Zwiebel   Zwiebel   Zucchini  Zucchini  Zucchini
Rosenkohl Rosenkohl Rosenkohl Paprika   Paprika
     Kohlrabi  Kohlrabi  Kartoffel Kartoffel Kartoffel Linse     Linse     Linse
Kartoffel Kartoffel Kartoffel Kohlrabi
bin: 1         1         1         1         1         0         1         0
1         1         0         0
binSum: 8
Path:
1,1,1,1,1,0,1,0,1,1,0,0 binSum: 8 Pattern: 1,1,1,1,1,1,1,1,1,1,1,1
1,1,1,1,1,0,0,1,1,1,0,0 binSum: 8 Pattern: 1,1,1,1,1,0,0
1,1,1,1,0,1,0,1,1,1,0,0 binSum: 8 Pattern: 1,1,1,1,0
1,1,1,0,1,1,0,0,1,1,0,0 binSum: 7 Pattern: 1,1,1,0
```

bauern4

```
Runtime: 0.052s
cultivation plan:
     Tomate    Tomate    Tomate    Paprika   Paprika   Paprika   Zwiebel   Zwiebel
Zwiebel   Zucchini  Zucchini  Zucchini
     Linsen    Kartoffel Kartoffel Kartoffel Kohlrabi  Kohlrabi  Kohlrabi
Kartoffel Kartoffel Kartoffel Linsen    Linsen
     Pastinake Pastinake Apfel     Apfel     Apfel     Banane    Banane    Banane
Paprika   Paprika   Paprika   Pastinake
bin: 1         1         1         1         1         0         1         1
1         1         0         0
binSum: 9
Path:
1,1,1,1,1,1,0,1,0,1,0,0 binSum: 8 Pattern: 1,1,1,1,1,1,1,1,1,1,1,1
1,1,1,1,1,1,0,0,1,1,0,0 binSum: 8 Pattern: 1,1,1,1,1,1,0,0
1,1,1,1,1,0,1,1,1,1,0,0 binSum: 9 Pattern: 1,1,1,1,1,0
```

bauern5

```
Runtime: 0.053s
cultivation plan:
     W W W J J J T T T J J J
     V C C C S S S N N N V V
     L L B B B U U U Z Z Z L
bin: 1 1 1 1 1 0 1 1 1 1 0 0
binSum: 9
Path:
1,1,1,1,1,0,1,1,1,1,0,0 binSum: 9 Pattern: 1,1,1,1,1,1,1,1,1,1,1,1
```

bauern6

```
Runtime: 0.186s
cultivation plan:
     P P P L L L B B B W W W
     M B B B H H H G G G M M
     V V A A A R R R Z Z Z V
bin: 1 1 1 1 1 1 1 1 1 1 0 0
binSum: 10
Path:
1,1,1,1,1,1,1,1,1,1,0,0 binSum: 10 Pattern: 1,1,1,1,1,1,1,1,1,1,1,1
```

6.2 Zusatzbeispiele

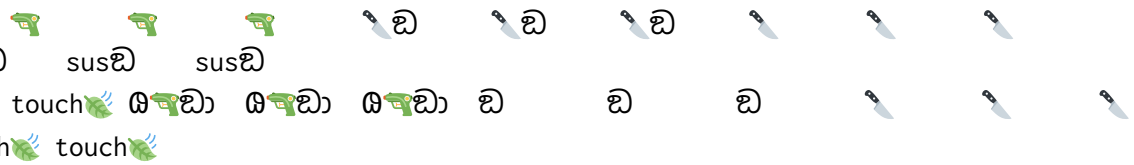
emoji

```
Runtime: 0.005s
cultivation plan:
```

```
bin: 1  1  1  1  0  0  0  0  0  0  0  0
binSum: 4
Path:
1,1,1,1,0,0,0,0,0,0,0,0 binSum: 4 Pattern: 1,1,1,1,1,1,1,1,1,1,1,1
```

amogus

```
Runtime: 0.085s
cultivation plan:
```



```
bin: 1       1       1       1       0       1       1       0       1       0
0       0
binSum: 7
```

```
Path:
1,1,1,1,0,1,1,0,1,0,0,0 binSum: 7 Pattern: 1,1,1,1,1,1,1,1,1,1,1,1
1,1,1,1,0,1,1,0,0,1,0,0 binSum: 7 Pattern: 1,1,1,1,0,1,1,0,0
1,1,1,1,0,1,0,1,0,1,0,0 binSum: 7 Pattern: 1,1,1,1,0,1,0
1,1,1,1,0,1,0,0,1,1,0,0 binSum: 7 Pattern: 1,1,1,1,0,1,0,0
1,1,1,1,0,0,1,1,0,1,0,0 binSum: 7 Pattern: 1,1,1,1,0,0
1,1,1,1,0,0,1,0,1,1,0,0 binSum: 7 Pattern: 1,1,1,1,0,0,1,0
1,1,1,0,1,1,0,1,0,1,0,0 binSum: 7 Pattern: 1,1,1,0
1,1,1,0,1,1,0,0,1,1,0,0 binSum: 7 Pattern: 1,1,1,0,1,1,0,0
1,1,1,0,1,0,1,1,0,1,0,0 binSum: 7 Pattern: 1,1,1,0,1,0
1,1,1,0,1,0,1,0,1,1,0,0 binSum: 7 Pattern: 1,1,1,0,1,0,1,0
1,1,0,1,1,1,1,0,0,1,0,0 binSum: 7 Pattern: 1,1,0
1,1,0,1,1,1,0,1,0,0,0,0 binSum: 6 Pattern: 1,1,0,1,1,1,0
1,1,0,1,1,0,1,0,1,1,0,0 binSum: 7 Pattern: 1,1,0,1,1,0
```

bauernt1

```
Runtime: 0.03s
cultivation plan:
     0 0 0 6 6 6 3 3 3 0 0 0
     0 1 1 1 0 0 0 4 4 4 0 0
     0 0 2 2 2 1 1 1 5 5 5 0
bin: 1 1 1 1 0 0 1 0 0 0 0 0
binSum: 5
Path:
1,1,1,1,0,0,1,0,0,0,0,0 binSum: 5 Pattern: 1,1,1,1,1,1,1,1,1,1,1,1
```

bauernt2

```
Runtime: 0.01s
cultivation plan:
     a a a d d d g g g j j j
     k b b b e e e h h h k k
     l l c c c f f f i i i l
bin: 1 1 1 1 1 1 1 1 1 1 1 1
binSum: 12
Path:
1,1,1,1,1,1,1,1,1,1,1,1 binSum: 12 Pattern: 1,1,1,1,1,1,1,1,1,1,1,1
```

bauernt3

```
Runtime: 0.005s
cultivation plan:
     0  0  0  3  3  3  6  6  6  9  9  9
     10 1  1  1  4  4  4  7  7  7  10 10
     11 11 2  2  2  5  5  5  8  8  8  11
bin: 1  0  0  1  0  0  1  0  0  1  0  0
binSum: 4
Path:
1,0,0,1,0,0,1,0,0,1,0,0 binSum: 4 Pattern: 1,1,1,1,1,1,1,1,1,1,1,1
```

7 Quelltext

script.js JavaScript

```
72  //Sortiere die Zutaten eines Gerichts
73  function eSort([a, b, c]) {
74   if (a > b) [a, b] = [b, a]
75   if (a > c) [a, c] = [c, a]
76   if (b > c) [b, c] = [c, b]
77   return [a, b, c]
78  }
79  //Berechne die ID eines Gerichts
80  let gId = ([a, b, c], z) => a + b * z + c * z ** 2
81  //Berechne die ID eines Essen
82  let eId = (e, z) => gId(eSort(e), z)
83  //Berechne, wie viele verschiedene Gerichte gekocht werden können
84  function seqUnique(seq, [kochbar, z], skip = 0) {
85   let essen = [, seq[skip], seq[1 + skip]]
86   let gerichte = []
87   let len = seq.length - skip
88   for (let i = 2; i < len || (len == 12 && i < len + 2); i++) {
89    essen = [essen[1], essen[2], seq[i % 12 + skip]]
90    gerichte.push(eId(essen, z))
91   }
92   gerichte = [...new Set(gerichte)]
93   let sum = 0
94   for (let i = 0; i < gerichte.length; i++) sum += kochbar.has(gerichte[i])
95   return sum
96  }
97  //Ermittle alle Paare von Essen, mit denen man potenziell 01, 1, 11 oder 111 ergänzen
    könnte
98  function getPairs(es, kochbar) {
99   let e = es.length
100  let pairs = [[], [], [], []]
101  let cond = [
102   (i, j) => es[i][2] == es[j][0],
103   (i, j) => es[i][1] == es[j][0] && es[i][2] == es[j][1],
104   (i, j) => cond[0](i, j) && seqUnique([...es[i], es[j][1], es[j][2]], kochbar) == 3,
105   (i, j) => seqUnique([...es[i], ...es[j]], kochbar) == 4
106  ]
107  for (let i = 0; i < e; i++) {
108   for (let j = 0; j < 4; j++) pairs[j][i] = []
109   for (let j = 0; j < i - i % 6; j++) for (let l = 0; l < 4; l++) if (cond[l](i, j)) {
110    pairs[l][i].push(j)
111    pairs[l][j ^ 1].push(i ^ 1)
112   }
113  }
114  return pairs
115 }
116 //Füge neue Essen hinzu, sodass nur neue Einsen erzeugt werden
117 function add1s(seqs, seq, [essen, edges, kochbar], len, repeat, prevUnique = -2) {
118  let unique = seqUnique(seq, kochbar, 1)
```

```
119   if (unique - prevUnique < len) return
120   if (unique >= seqs[0]) {
121    if (unique > seqs[0]) {
122     seqs.length = 1
123     seqs[0] = unique
124    }
125    seqs.push(seq)
126   }
127   if (seq.length > 13 - len || (prevUnique > -2 && !repeat)) return
128   let next = edges[len][seq[0]]
129   for (let i = 0; i < next.length; i++) {
130    add1s(seqs, [next[i], ...seq.slice(1), ...essen[next[i]].slice(3 - len, 3)], [essen,
      edges, kochbar], len, repeat, unique)
131   }
132  }
133  //Füge neue Essen hinzu, sodass auch Nullen entstehen
134  function add0s(seqs, seq, [essen, edges, kochbar], len) {
135   let prevUnique = seqUnique(seq, kochbar, 1)
136   if (prevUnique == seqs[0]) seqs.push(seq)
137   if (seq.length > 12 - len) return
138   let next = edges[0][seq[0]]
139   if (len == 2) {
140    next = []
141    for (let i = 0; i < essen.length; i++) next[i] = i
142   }
143   for (let i = 0; i < next.length; i++) {
144    let newSeq = [next[i], ...seq.slice(1), ...essen[next[i]].slice(2 - len, 3)]
145    let unique = seqUnique(newSeq, kochbar, 1)
146    if (unique == prevUnique + 1 && seqUnique(newSeq.slice(0, -1), kochbar, 1) == prevUnique)
      {
147     if (unique > seqs[0]) {
148      seqs.length = 1
149      seqs[0] = unique
150     }
151     seqs.push(newSeq)
152    }
153   }
154  }
155  //Füge das nächste Essen hinzu
156  function add(bin, len, seqs, data, repeat = 1) {
157   let newSeqs = [seqs[0]]
158   if (bin == 1) {
159    for (let i = 1; i < seqs.length; i++) add1s(newSeqs, seqs[i], data, len, repeat)
160   } else {
161    for (let i = 1; i < seqs.length; i++) add0s(newSeqs, seqs[i], data, len)
162   }
163   return newSeqs
164  }
165  //Ermittle die lexikografisch größten Anbaupläne aus unfertigen Anbauplänen
166  function lexMax(seqs, data) {
167   let oldUnique = 0
```

```
168   while (seqs[0] > oldUnique) {
169    oldUnique = seqs[0]
170    for (let i = 3; i > 0; i--) seqs = add(1, i, seqs, data)
171    let prevUnique = seqs[0]
172    seqs = add(0, 1, seqs, data)
173    if (prevUnique == seqs[0]) seqs = add(0, 2, seqs, data)
174   }
175   for (let i = 1; i < seqs.length; i++) seqs[i].push(...Array(13 - seqs[i].length).fill(0))
176   return seqs
177  }
178  //Ermittle den Anfang der lexikografisch größtmöglichen Lösung, die kleiner als die
     Obergrenze ist
179  function patternMax(seqs, bin, data) {
180   let ind = 1
181   let repeats = 0
182   while (seqs[1].length - 3 < bin.length && repeats < 3) {
183    ind = seqs[1].length - 3
184    if (bin[ind]) {
185     seqs = add(1, 1 + (bin[ind + 1] || 0) + ((bin[ind + 2] || 0) && (bin[ind + 1] || 0)) -
     repeats, seqs, data, 0)
186    } else {
187     seqs = add(0, 1 + (!bin[ind + 1] && ind + 1 < bin.length), seqs, data, 0)
188    }
189    repeats = ind == seqs[1].length - 3 ? repeats + 1 : 0
190   }
191   return seqs
192  }
193  //Berechne den Binary-String eines Anbauplans
194  function getBin(seq, kochbar) {
195   let bin = []
196   let prevUnique = 0
197   let es = [seq[1], seq[2], seq[3]]
198   for (let i = 0; i < 12; i++) {
199    prevUnique += bin[i] = +(seqUnique(es, kochbar) > prevUnique)
200    es.push(seq[(i + 3) % 12 + 1])
201   }
202   return bin
203  }
204  //Prüfe, ob im Binary-String drei aufeinanderfolgende Nullen vorkommen
205  function is000(bin) {
206   let streak = 0
207   for (let i = 0; i < bin.length; i++) if ((streak = streak * !bin[i] + !bin[i]) == 3 && i <
     10) return 1
208  }
209  //Zähle die Anzahl der Einsen im Binary-String
210  function binSum(bin) {
211   let sum = 0
212   for (let i = 0; i < bin.length; i++) sum += bin[i]
213   return sum
214  }
215  //Teste, ob dieser Binary-String erfüllbar und besser wäre
```

```
function testBin(bin, sum) {
 if (is000(bin)) return
 let len = bin.length
 bin[11] = 0
 let streak1 = 0
 while (bin[streak1]) streak1++
 let streak0 = 1 + (!bin[streak1 + 1] && streak1 + 1 < len)
 for (let i = 0; i < 11 - len; i++) bin[len + i] = +(i % (streak1 + streak0) < streak1)
 if (binSum(bin) <= sum) return
 bin.length = len
 return bin
}
//Ermittle die nächstkleinere Möglichkeit für einen besseren Anbauplan
function nextBin(bin, bestSum) {
 for (let i = 11; i > 0; i--) if (bin[i]) {
  let res = testBin([...bin.slice(0, i), 0], bestSum)
  if (res) return res
 }
}
//Erstelle alle möglichen Anbaupläne für das erste Essen
function startSeqs(essen) {
 let seqs = [1]
 for (let i = 0; i < essen.length; i++) seqs[i + 1] = [i, ...essen[i]]
 return seqs
}
//Ermittle den optimalen Anbauplan
function main([g, zutaten, essen, kochbar]) {
 //Ermittle die lexikografisch größte Lösung
 let edges = getPairs(essen, kochbar)
 let edgeSum = 0
 for (let i = 0; i < edges[0].length; i++) edgeSum += edges[0][i].length
 let data = [essen, edges, kochbar]
 let addE = i => (essen[i * 6] || [0, 0, 0])
 let seqOpt = [Math.min(4, essen.length / 6),
 [0, ...addE(0), ...addE(1), ...addE(2), ...addE(3)]]
 let seqs = edgeSum ? lexMax(startSeqs(essen), data) : seqOpt
 let bin = getBin(seqs[1], kochbar)
 let path = `\n${bin} binSum: ${seqs[0]} Pattern: 1,1,1,1,1,1,1,1,1,1,1,1`
 let newBin
 let bestSeqs = seqs
 //Suche nach besseren Anbauplänen, bis keine Verbesserung mehr möglich ist
 while (seqs[0] < 10 && seqs[0] < g && (newBin = nextBin(bin, bestSeqs[0]))) {
  seqs = patternMax(startSeqs(essen), newBin, data)
  if (seqs[1].length - 3 < newBin.length) {
   let prevUnique = seqs[0]
   seqs = add(0, 1, seqs, data)
   if (prevUnique == seqs[0]) seqs = add(0, 2, seqs, data)
  }
  seqs = lexMax(seqs, data)
  bin = getBin(seqs[1], kochbar)
```

```
    if (seqs[0] > bestSeqs[0]) bestSeqs = seqs
    path += `\n${bin} binSum: ${seqs[0]} Pattern: ${newBin}`
  }
  return [bestSeqs, zutaten, kochbar, path]
}
```

main.py — Python

```python
import sys
from time import time_ns

sys.set_int_max_str_digits(10000)


def comb(a, b, c):
    global n
    return a * (n ** 2) + b * n + c


def decomb(i):
    global n
    return i // (n ** 2), (i % (n ** 2)) // n, i % n


def recomb(i):
    global n
    return comb(*sorted(list(decomb(i))))


debug = False
months = 12
path = 'bauern2.txt'
g = []
rg = {}
start = time_ns()
with open(path, encoding='utf-8') as f:
    n = int(f.readline())
    for k in range(n):
        g.append(f.readline().rstrip())
        rg[g[-1]] = k
    d = [0 for _ in range(n ** 3 + 1)]
    m = int(f.readline())
    for k in range(m):
        a, b, c = tuple(f.readline().rstrip().split())
        a = int(rg[a])
        b = int(rg[b])
        c = int(rg[c])
        d[(n ** 2) * a + n * b + c] += 1
        d[(n ** 2) * a + n * c + b] += 1
        d[(n ** 2) * b + n * a + c] += 1
        d[(n ** 2) * b + n * c + a] += 1
```

```
        d[(n ** 2) * c + n * a + b] += 1
        d[(n ** 2) * c + n * b + a] += 1
dp = [[(0, set(), 0) for _ in range(n ** 3)] for _ in range(months + 1)]
for j in range(n ** 3):
    if d[j]:
        dp[1][j] = (d[j], {recomb(j)}, (j // (n ** 2), (j % (n ** 2)) // n))
for i in range(2, months - 1):
    for j in range(n ** 3):
        if dp[i - 1][j][0] == 0:
            continue
        if type(dp[i - 1][j][1]) is not set:
            print(type(dp[i - 1][j][1]), decomb(j))
            raise ValueError
        for k in range(n):
            b = (j % (n ** 2)) // n
            c = j % n
            if k == b or k == c:
                continue
            target = comb(b, c, k)
            rtarget = recomb(target)
            if dp[i][target][0] < dp[i - 1][j][0] + d[target] * (rtarget not in dp[i - 1][j][1]):
                dp[i][target] = (dp[i - 1][j][0] + d[target] * (rtarget not in dp[i - 1][j][1]),
                                 dp[i - 1][j][1].copy() | {rtarget}, dp[i - 1][j][2])

if debug:
    for i in range(len(dp[months - 2])):
        if dp[months - 2][i][0] == 10:
            print(decomb(i), dp[months - 2][i][0], *dp[months - 2][i][2])
            #print(bool(dp[months - 2][i][1] & (1 << recomb(comb(decomb(i)[1], decomb(i)[2], dp[months - 2][i][2][0])))),
            #      (decomb(recomb(comb(decomb(i)[1], decomb(i)[2], dp[months - 2][i][2][0])))))  # obsolete

for j in range(n ** 3):
    if dp[months - 2][j][0] == 0:
        continue
    oa, ob = dp[months - 2][j][2]
    ta, tb, tc = decomb(j)
    nt = comb(tb, tc, oa)
    rnt = recomb(nt)

    if debug:
        if dp[months - 2][j][0] == 10:
            print('a', decomb(j))
            print('b', decomb(rnt), decomb(nt))

    if dp[months - 1][nt][0] < dp[months - 2][j][0] + d[nt] * (rnt not in dp[months - 2][j][1]):
```

```
89          dp[months - 1][nt] = (dp[months - 2][j][0] + d[nt] * (rnt not in dp[months - 2][j]
            [1]),
90                                dp[months - 2][j][1].copy() | {rnt}, dp[months - 2][j][2])
91          nnt = comb(tc, oa, ob)
92          rnnt = recomb(nnt)
93
94          if dp[months][nnt][0] < dp[months - 1][nt][0] + d[nnt] * (rnnt not in dp[months - 1]
            [nt][1]):
95              dp[months][nnt] = (dp[months - 1][nt][0] + d[nnt] * (rnnt not in dp[months - 1]
                [nt][1]),
96                                 dp[months - 1][nt][1].copy() | {rnnt}, dp[months - 1][nt][2])
97
98  print('Max. ' + str(max(i[0] for i in dp[months])) + ' dishes are cookable yearly')
99  print('Finished in ' + str((time_ns() - start) / 1e6) + ' ms')
```