Aufgabe 3: Hex-Max

Teilnahme-ID: 62048

Bearbeiter dieser Aufgabe:
Leandro Conte

14. April 2022

Inhaltsverzeichnis

1 Lösungsidee

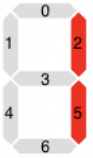
1.1 Das Problem

Gegeben ist ein String α aus N Hexadezimalziffern, wobei jeder Ziffer eine einzigartige Kombination aus bis zu sieben Segmenten zugewiesen wird. Durch das 'Umlegen' dieser Segmente, welches maximal M erfolgen darf, soll nun der größtmöglichste Hexadezimalstring gebildet werden.
Der Lösungsidee beruht auf folgender Abstraktion des String-Umformungsvorgangs:

1.2 Wie man sich die schrittweise Umformung des Strings vorstellen kann

Um eine Ziffer in eine andere zu transformieren müssen Segmente umgelegt werden. Dabei gibt es für zwei Ziffern x und y, Segmente, die in x sind, wo sie in y nicht mehr sein dürfen, und Segmente, die in y sind, ohne in x zu sein.

$$b_{x\to y} := (\text{Anzahl der Segmentpositionen, die nur in x belegt sind,}$$
$$\text{Anzahl der Segmentpositionen, die nur in y belegt sind})$$
$$\implies b_{B\to D} = (1,1)$$

An dieser Stelle muss hinzugefügt werden, dass die Segmentdarstellung der Ziffern, auch als Binärdarstellung mit 7-bits, gedacht werden kann. Ordnet man die Segmentplätze wie in Abbildung 1 

, kann jeder Ziffer eine 7 bit Zahl zugeordnet werden. $B \to D$ entspricht dann $0101111 \to 0011111$ (0 vorne)
Solange die zwei Komponenten von $b_{x\to y}$ gleich sind, intern also an genauso vielen Positionen Segmente entfernt, wie hinzugefügt werden müssen, kann die Transformation ohne Beachtung des Rest des Strings erfolgen. Sie kostet dann $b_{x\to y}[0]$ Umlegungen.
Es gibt aber auch Fälle, wie zum Beispiel $b_{2\to 7} = (2,1)$, in denen die Segmentanzahlen der zwei Ziffern nicht gleich sind. Solche Transformationen sind immer noch möglich, da Transformationen mehrerer Ziffern eines Strings sich ausbalancieren können. Beispiel:
“$2F$“ $\to$ “$C0$“, wobei $b_{2\to C} = (3,1)$ und $b_{F\to 0} = (1,3)$
Die erste Transformation muss zwei Segmente nach außen abgeben. Sie kann trotzdem stattfinden, da gleichzeitig die zweite Transformation zwei Segmente von außen aufnehmen muss.
Man kann sich für einen String γ während des Transformationsvorgangs eine zusäzliche Information g, das Gleichgewicht des Strings, bzw. die Differenz an Segmenten zum Originalstring, vorstellen. Dieses ist anfänglich 0, wird jedoch von Transformationen verändert. Was das heißen soll kann man am Beispiel erkennen:

$$\gamma = \text{“}2F\text{“} \quad g = 0$$
$$\xrightarrow{2\to C} \gamma = \text{“}CF\text{“} \quad g = 2$$
$$\xrightarrow{F\to 0} \gamma = \text{“}C0\text{“} \quad g = 0$$

Solange das vorgestellte Gleichgewicht nicht null ist, ist der String ungültig, da er teilweise aus nichtexistenten Segmenten besteht, oder Segmente ins Nichts abgelegt wurden. Eine Transformation $x \to y$ beeinflusst das Gleichgewicht des Systems folgendermaßen:

$$b_{x\to y} = (w,z) \implies g_{x\to y} = w - z$$

Man kann sich g als Ablage vorstellen, die positiv ist, wenn der String weniger Segmente als zuvor besitzt. Kann man einer Transformationen Umlegungskosten zuweisen? Die Transformation $x \to y$ kostet zuerst einmal die inneren Kosten, also $\min(w,z)$. Die Umlegungen, die in ”Zusammenarbeit” mit einer anderen Transformation stattfindeen, werden nur halb gezählt, da sie eben bei beiden Transformationen betrachtet werden. Für die Kosten gilt also:

$$k_{x\to y} = \min(w,z) + \frac{1}{2}|w - z| \quad (\text{Immer mit } (w,z) = b_{x\to y})$$

Es muss an dieser Stelle hinzugefügt werden, dass das gerade erklärte System mit den Transformationen, der vorgestellten ”Segmentbank” und den rationalen Kosten, nicht beschreibt wie am Schluss die Segmente tatsächlich umgelegt werden. Es wird nur im Verfahren den Ergebnisstring zu finden benötigt.
Damit ein String nach mehrern Transformationen, die dazu führen, dass $g = 0 \wedge k \leq M$, auch im Sinne der Aufgabenstellung gültig ist muss noch folgendes gezeigt werden.

1.3 Weshalb eine Ziffer nie komplett leer ist

Wird ein String in einen String mit gleicher Anzahl an Segmenten umgeformt, dann ist diese Umformung möglich ohne, dass je eine Ziffer komplett leer ist. Folgende Umlegungsstrategie sorgt dafür:
In einem ersten Durchgang merkt man sich alle Segmentpositionen die verändert werden müssen, dann verändert man alle gegensätzlichen Paare, also entfernen - hinzufügen, die in der gleichen Ziffer liegen. Nun ist immernoch keine Ziffer leer. Dann löst man alle anderen Paare auf, wobei auch hier eine Ziffer nie leer wird, da die bei der internen Auflösung bewegten Segmente an ihren Positionen bleiben, und die Segmente die überhaupt nicht bewegt werden müssen auch belegt bleiben.

1.4 Finden der größten Hexzahl

Der Lösungsstring wird Ziffer für Ziffer zusammengesetzt. Damit der resultierende String so groß wie möglich ist werden in absteigender Reihenfolge die möglichen bedeutendsten Ziffern getestet. Der allererste Schritt für $\alpha =$ "$D24$" ist also zu bestimmen, was die minimalen Kosten für "$D24$" $\rightarrow$ "$F--$" sind. Der Suffix ist unbestimmt, da seine lexikographische Größe weniger wichtig als die der bedeutendsten Stelle ist. $D \rightarrow F$ resultiert in g = 1 für den Suffix. Dieser Suffix muss also ausgehend von g=1 "balanciert"werden, also mit minimalen Kosten so umgeformt werden, dass g zu 0 wird. Die minimalen Kosten für die balancierung des Suffixes seien $f(i, g)$ wobei i die bedeutendste Stelle des Suffixes ist und g das Ungleichgewicht. Wenn $k_{D\rightarrow F} + f(1, 1) \leq M$ wäre, könnte man jetzt die 0. Stelle der Lösung auf F festlegen, doch im Beispiel ist das nicht der Fall. Also geht man weiter zu Umformung in die nächstkleinere Ziffer E und so weiter. Hat man die bedeutendste Ziffer bestimmt, geht man zu nächstbedeutendsten über.
Zusammengefasst wird die Lösung also Ziffer für Ziffer, beginnend bei der bedeutendsten Stelle, zusammengesetzt. Ausgehend von den Kosten für die bisherigen Transformationen und dem aus diesen resultierten Ungleichgewicht, wird die aktuelle Stelle in die größtmöglichste Ziffer transformiert. Dass die Kosten am Schluss nicht M übersteigen und dass der resultierende String gleich viele Segmente wie das Original enthält, wird von f garantiert. Da jedesmal die größtemöglichste Ziffer gewählt wird, ist auch das Ergebnis maximal.

1.5 f(i, g)

$$f(N, g) = \begin{cases} 0, \text{ falls } g = 0 \\ \infty, \text{ ansonsten} \end{cases} \qquad \text{0-basierte Indexe}$$

$$f(i, g) = \min\left(\infty, \min_{y \in Z}\left(k_{x\rightarrow y} + f(i + 1, g + g_{x\rightarrow y})\right)\right)$$

Z ist die Menge der Ziffern $\quad x$ ist die Ziffer im Original

f versucht an jeder Stelle alle möglichen Ziffern einzusetzen, un gibt dann die minimalen Kosten zurück. Da die Definitionsmenge von f relativ begrenzt ist, kann man mit dynamic programming die Ergebnisse memoisieren. Der Speicherbedarf ist dann $O(N^2)$ und die Laufzeit $O(N^2 \cdot |Z|)$.

1.6 Zusammenfassung und Laufzeit

Für den gesamten Algorithmus ergibt sich somit für das Finden des Lösungsstrings die Laufzeit $O(N \cdot |Z| + N^2 \cdot |Z|)$, da die Werte von f gespeichert werden. Für die Hexadezimalziffern ist die Laufzeit also $O(N^2)$. Hat man einmal den Lösungsstring kann man mit der in 1.3 vorgestellten Strategie in linearer Laufzeit die Umlegungsschritte berechnen. BILD EINFÜGEN!

2 Umsetzung

Die Umsetzung ist in Rust geschrieben und beinhaltet die "serdejson" und "rand" Pakete.

2.1 I/O

Eingabe und Ausgabe Funktionen sind in "io.rs" enthalten. Die structs *TInput* und *TOutput* handhaben das Einlesen der Problemstellungen und die Ausgabe der Lösungen. Um verschiedene Ziffersysteme zu erlauben, liest der *Characters* struct die Informationen zu den verfügbaren Ziffern aus einer json Datei. Die "chars.json" Datei enthält das verwendete Hexadezimalsystem. In der Datei wird jeder Ziffer ein Bitstring zugewiesen, der die Segmente beschreibt. Während der Verarbeitung werden Objekte der *Character* Klasse verwendet. Diese werden auch von *Characters* bereitgestellt.

2.2 Hauptteil

Die Verarbeitungslogik befindet sich in "processing.rs". Hier wird *Characters* um mehrere wichtige Methoden erweitert.

```
pub fn transform_balance(&self, a: &Character, b: &Character) -> (u64, u64)
```

Vergleicht die bitstrings von a und b und entspricht $b_{a \to b}$

```
pub fn transform_effect(&self, a: &Character, b: &Character) -> (f64, i64)
```

Gibt in einem Tupel (Kosten, Gleichgewichtsänderung) einer Transformation zurück.
Während der Verarbeitung werden Strings oft als Vektoren der *Character* Objekte dargestellt und die *stovec* Methode macht diese Konversion. Zuletzt gibt es noch:

```
pub fn string_steps(&self, a: &[&Character], b: &[&Character]) -> Vec<Step>
```

Diese Methode iteriert ähnlich wie in 1.3 über die Segmentpositionen und bestimmt so eine gültige Sammlung an Umlegungen, die a in b umformen. Die *Step* Objekte enthalten zwei Segmentpositionen und den Zustand nach der Umformungen als Vektor von Bitmasken.
Die Verarbeitung findet mittels der *process* Funktion statt, die die Eingabe, sowie das Ziffernsystem erhält und den in 1.4 beschriebenen Algorithmus implementiert. Optional kann der Ausgabe am Schluss der Vektor mit den Umlegungen hinzugefügt werden. Der Kern von *process* ist folgender

```
for i in 0..n {
        for c in chars.chars.iter().rev() { //Reverse to iterate in descending order
            let effect = chars.transform_effect(context.s[i], c);
            let nbal = cbal + effect.1;
            let ncost = cost + effect.0;
            if ncost+balancing_cost(&mut context, i+1, nbal) <= input.m as f64 {
                n_string.push(c);
                cost = ncost;
                cbal = nbal;
                break;
}}}
```

Die *balancing_cost* Funktion entspricht hier $f(i, g)$. Mit *context* erhält sie zusätzlich das dparray und weitere Informationen. Der Kern ist hier:

```
let mut cmin: f64 = INFINITY;
    for c in ctx.chars.chars.iter().rev() {
        let effect = ctx.chars.transform_effect(ctx.s[i], c);
        cmin = f64::min(cmin, balancing_cost(ctx, i+1, bal+effect.1)+effect.0);
    }
```

2.3 Verifikation

Obwohl das Verfahren theoretisch einwandfrei sein müsste, enthält die Umsetzung mehrere Methoden, die die Korrektheit der Ausgabe überprüfen. Die *string_cost* Methode von *Characters* prüft wie viele Umlegungen für die Umformung von einem String in den anderen notwendig sind und, ob die zwei Strings überhaupt gleich viele Segmente besitzen. In der Datei "testing.rs" ist neben einer Funktion

`run_samples`

, die alle Beispiele der Bwinf-Seite verarbeitet und die Resultate speichert auch eine Funktion

`run_randomized`

, die zufällig generierte Eingaben verarbeitet.

2.4 Visualisation

Das Ausgabe Format ist:
<Lösungsstring>
<Benötigte zeit in ms>
(<von Charakter>, <von Segment>) (<zu Charakter>, <zu Segment>) <Maske erster Ziffer> \
<Masker zweiter Ziffer> ...
<weiter die Umlegungen beschreibende Zeilen>
Um die Umlegungen zu visualisieren gibt es ein Python script: visu.py

3 Erweiterungen

3.1 Verschiedene Ziffersysteme

3.2 Parallelisierung

4 Beispiele

Beispiel 0
Eingabe:*hexmax0.txt*
Ausgabe:

```
EE4
0
(-1, -1) (-1, -1) 124 93 46
(0, 5) (0, 1) 94 93 46
(0, 2) (0, 0) 91 93 46
(1, 2) (1, 1) 91 91 46
```

Beispiel 1
Eingabe:*hexmax1.txt*
Ausgabe:

```
FFFEA97B55
0
(-1, -1) (-1, -1) 107 119 111 83 46 109 36 122 107 107
(0, 6) (0, 4) 59 119 111 83 46 109 36 122 107 107
(1, 6) (1, 3) 59 63 111 83 46 109 36 122 107 107
(2, 6) (2, 4) 59 63 63 83 46 109 36 122 107 107
(2, 5) (3, 3) 59 63 31 91 46 109 36 122 107 107
(2, 2) (4, 4) 59 63 27 91 62 109 36 122 107 107
(1, 5) (4, 0) 59 31 27 91 63 109 36 122 107 107
(1, 2) (5, 1) 59 27 27 91 63 111 36 122 107 107
(0, 5) (6, 0) 27 27 27 91 63 111 37 122 107 107
```

Beispiel 2
Eingabe:*hexmax2.txt*
Ausgabe:

```
FFFFFFFFFFFFFFFFD9A9BEAEE8EDA8BDA989D9F8
2
(-1, -1) (-1, -1) 123 109 93 122 93 111 122 109 127 27 36 36 127 46 111 119 36 107 63 109 122 83 63 9
(1, 6) (1, 4) 123 61 93 122 93 111 122 109 127 27 36 36 127 46 111 119 36 107 63 109 122 83 63 91 91
(1, 5) (1, 1) 123 31 93 122 93 111 122 109 127 27 36 36 127 46 111 119 36 107 63 109 122 83 63 91 91
(2, 6) (2, 1) 123 31 31 122 93 111 122 109 127 27 36 36 127 46 111 119 36 107 63 109 122 83 63 91 91
(3, 6) (3, 0) 123 31 31 59 93 111 122 109 127 27 36 36 127 46 111 119 36 107 63 109 122 83 63 91 91 9
(4, 6) (4, 1) 123 31 31 59 31 111 122 109 127 27 36 36 127 46 111 119 36 107 63 109 122 83 63 91 91 9
(5, 6) (5, 4) 123 31 31 59 31 63 122 109 127 27 36 36 127 46 111 119 36 107 63 109 122 83 63 91 91 93
(6, 6) (6, 0) 123 31 31 59 31 63 59 109 127 27 36 36 127 46 111 119 36 107 63 109 122 83 63 91 91 93
(7, 6) (7, 4) 123 31 31 59 31 63 59 61 127 27 36 36 127 46 111 119 36 107 63 109 122 83 63 91 91 93 8
(7, 5) (7, 1) 123 31 31 59 31 63 59 31 127 27 36 36 127 46 111 119 36 107 63 109 122 83 63 91 91 93 8
(10, 5) (10, 4) 123 31 31 59 31 63 59 31 127 27 20 36 127 46 111 119 36 107 63 109 122 83 63 91 91 93
(10, 2) (10, 3) 123 31 31 59 31 63 59 31 127 27 24 36 127 46 111 119 36 107 63 109 122 83 63 91 91 93
(8, 6) (10, 1) 123 31 31 59 31 63 59 31 63 27 26 36 127 46 111 119 36 107 63 109 122 83 63 91 91 93 8
(8, 5) (10, 0) 123 31 31 59 31 63 59 31 31 27 27 36 127 46 111 119 36 107 63 109 122 83 63 91 91 93 8
(11, 5) (11, 4) 123 31 31 59 31 63 59 31 31 27 27 20 127 46 111 119 36 107 63 109 122 83 63 91 91 93
(11, 2) (11, 3) 123 31 31 59 31 63 59 31 31 27 27 24 127 46 111 119 36 107 63 109 122 83 63 91 91 93
(8, 2) (11, 1) 123 31 31 59 31 63 59 31 27 27 27 26 127 46 111 119 36 107 63 109 122 83 63 91 91 93 8
(7, 2) (11, 0) 123 31 31 59 31 63 59 27 27 27 27 27 127 46 111 119 36 107 63 109 122 83 63 91 91 93 8
(13, 5) (13, 4) 123 31 31 59 31 63 59 27 27 27 27 27 127 30 111 119 36 107 63 109 122 83 63 91 91 93
(13, 2) (13, 0) 123 31 31 59 31 63 59 27 27 27 27 27 127 27 111 119 36 107 63 109 122 83 63 91 91 93
```

```
(14, 6) (14, 4) 123 31 31 59 31 63 59 27 27 27 27 27 127 27 63 119 36 107 63 109 122 83 63 91 91 93 8
(15, 6) (15, 3) 123 31 31 59 31 63 59 27 27 27 27 27 127 27 63 63 36 107 63 109 122 83 63 91 91 93 83
(15, 5) (16, 6) 123 31 31 59 31 63 59 27 27 27 27 27 127 27 63 31 100 107 63 109 122 83 63 91 91 93 8
(15, 2) (16, 4) 123 31 31 59 31 63 59 27 27 27 27 27 127 27 63 27 116 107 63 109 122 83 63 91 91 93 8
(14, 5) (16, 3) 123 31 31 59 31 63 59 27 27 27 27 27 127 27 31 27 124 107 63 109 122 83 63 91 91 93 8
(14, 2) (17, 2) 123 31 31 59 31 63 59 27 27 27 27 27 127 27 27 27 124 111 63 109 122 83 63 91 91 93 8
(12, 6) (19, 1) 123 31 31 59 31 63 59 27 27 27 27 27 63 27 27 27 124 111 63 111 122 83 63 91 91 93 83
(12, 5) (21, 3) 123 31 31 59 31 63 59 27 27 27 27 27 31 27 27 27 124 111 63 111 122 91 63 91 91 93 83
(12, 2) (25, 5) 123 31 31 59 31 63 59 27 27 27 27 27 27 27 27 27 124 111 63 111 122 91 63 91 91 125 8
(6, 5) (25, 1) 123 31 31 59 31 63 27 27 27 27 27 27 27 27 27 27 124 111 63 111 122 91 63 91 91 127 83
(5, 5) (26, 3) 123 31 31 59 31 31 27 27 27 27 27 27 27 27 27 27 124 111 63 111 122 91 63 91 91 127 91
(5, 2) (29, 3) 123 31 31 59 31 27 27 27 27 27 27 27 27 27 27 27 124 111 63 111 122 91 63 91 91 127 91
(4, 2) (32, 4) 123 31 31 59 27 27 27 27 27 27 27 27 27 27 27 27 124 111 63 111 122 91 63 91 91 127 91
(3, 5) (32, 0) 123 31 31 27 27 27 27 27 27 27 27 27 27 27 27 27 124 111 63 111 122 91 63 91 91 127 91
(2, 2) (34, 2) 123 31 27 27 27 27 27 27 27 27 27 27 27 27 27 27 124 111 63 111 122 91 63 91 91 127 91
(1, 2) (36, 6) 123 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 124 111 63 111 122 91 63 91 91 127 91
(0, 6) (36, 4) 59 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 124 111 63 111 122 91 63 91 91 127 91
(0, 5) (36, 3) 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 124 111 63 111 122 91 63 91 91 127 91
```

Beispiel 3
Eingabe:*hexmax3.txt*
Ausgabe:

```
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFAA98BB8B9DFAFEAE888DD888AD8BA8EA8888
11
```

Beispiel 4
Eingabe:*hexmax4.txt*
Ausgabe:

```
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEB8DE88BAA8ADD888898E9BA88AD98988F898AB7AF7BDA8A61BA7D4AD8F888
9
```

Beispiel 5
Eingabe:*hexmax5.txt*
Ausgabe:

```
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
↪ FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
↪ FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
↪ FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
↪ FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
↪ FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
↪ FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
↪ FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
↪ FFFFFFFFFFFFFFFFFFF88EFA9EBE89EFA99FBDAA8E8EAD88AB899F8E8F9AA9E9AD88988EDA9A99888
↪ EDAD989A8BAFD8A888888888888888888888888888888888888888888888888888888888888888888
↪ 888888888888888888888888888888888888888888888888888888888888888888888888888888888
↪ 888888888888888888888888888888888888888888888888888888888888888888888888888888888
↪ 888888888888888888888888
414
```

EXTRA BEISPIELE!!e

5 Quellcode

TODO!