Training für Programmierwettbewerbe - Exam Questions

Hanno Barschel (174761)

1 Introduction

(1) Zeit Komplexität von drei Funktionen:

- **foo(.)** hat eine Zeitkomplexität von $\mathcal{O}(n)$ (wobei n die Länge des gegebenen Vektors ist).

  Das folgt direkt daraus, dass wir genau 2 Vorschleifen die jeweils den gesamten Vektor durchgehen haben also $\mathcal{O}(2n) = \mathcal{O}(n)$.

- **print_ pairs(.)** hat eine Zeitkomplexität von $\mathcal{O}(n^2)$.

  Für jedes Element des Vektors ruft die Funktion nochmal jedes Element des Vektors auf und printed beide als Paar aus. Somit kommt eine Zeitkomplexität von $\mathcal{O}(n^2)$ zustande.

- **print_ unordered_ pairs(.)** hat eine Laufzeitkomplexität von $\mathcal{O}(n^2)$.

  Für jedes Element des Vektors rufen wir alle Folgeelemente auf. Das sind es $n - (i + 1)$ Aufrufe für Element i (i startet bei 0). Damit kommen wir auf eine Laufzeit von

  $$\sum_{i=0}^{n-1}(n - (i + 1)) = n^2 - \sum_{i=1}^{n} i = n^2 - \frac{n \cdot (n + 1)}{2} = n^2 - \frac{n^2}{2} - \frac{1}{2} \in \mathcal{O}(n^2).$$

- **all_ fib(.)** hat eine Zeitkomplexität von $\mathcal{O}(n)$.

  Die for-Schleife printed uns im Prinzip alle Fibonacci Zahlen bis n aus. Sobald wir eine Null in $memo[i]$ finden sind aber bereits $memo[i-1]$ und $memo[i-2]$ berechnet (für ein beliebiges i größer als 4, für i=0,1 gibt die Funktion per Definition das Ergebnis direkt zurück), somit müssen wir diese nur addieren und die Rekursionstiefe ist somit durch 1 beschränkt.

(2) Algorithmen Beispiele:

- $\mathcal{O}(1)$: Für eine gegebene Hashtabelle und einen Index gibt der Algorithmus den Wert für den Index zurück.
- $\mathcal{O}(\log(n))$: In einer sortierten Liste ein gegebenes Element suchen.
- $\mathcal{O}(n)$: Der Algorithmus berechnet zu einem gegeben Array die Summe aller Elemente und gibt diese zurück.
- $\mathcal{O}(n \cdot \log(n))$: Ein guter Sortier Algorithmus (Merge Sort).
- $\mathcal{O}(n^2)$: Quicksort (worst-case Analyse).

(3) Ersteinmal natürlich ordentlich zuhören wenn die Frage gestellt ist. Dabei sollten wir uns alles merken und auch auf Besonderheiten achten. Dann wählen wir ein geeignetes Beispiel. Dabei sollten wir immer im Hintergrund behalten, dass oft Spezialfälle besonders zu betrachten sind. Anschließend schreiben wir einen brute force Algorithmus auf und betrachten dessen Zeit sowie Platzkomplexität. Jetzt optimieren wir diesen noch indem wir das Ganze möglichst einfach machen, überflüssige und doppelte Arbeit vermeiden. Mit Hilfe dieser Betrachtungen verbessern wir den Brutforce Algorithmus schließlich noch.

(4) Gegeben sei eine Liste mit Natürlichen Zahlen. Gesucht sei die maximale Differenz zweier Werte im Array wobei der vordere (steht weiter vorne im Array) Wert kleiner als der hintere sein muss. Wir iterieren mittels zweier indexe, tmp_min und tmp die am Anfang beide auf 0 gesetzt werden. Gleichzeitig haben wir die Variable $max\_diff = 0$ in der die bisher gefundene maximale Differenz gespeichert wird.

Nun beginnen wir mittels einer for-Schleife mit Variablen tmp über den Array zu iterieren: Ist der Wert im Array an Stelle tmp kleiner oder gleich als der an der Stelle tmp_min so setzen wir tmp_min auf tmp. Ist tmp größer so berechnen wir die Differenz zwischen tmp und tmp_min. Ist diese größer als max_diff so ersetzen wir diese, ansonsten inkrementieren wir tmp einfach weiter.

(5) Ich würde es mit einer Hashmap checken: Wir testen erst einmal ob die Strings die gleiche Länge haben (wenn nicht dann return 0). Danach erstellen wir 2 Hashmaps (für jedes Wort eine). Anschließend gehen wir mittels einer for-Schleife beide strings durch und für jeden Buchstaben inkrementieren wir den zugehörigen Hashwert. Abschließend vergleichen wir beide Hashmaps.

$return(hashmap\_string1 == hashmap\_string2)$.

2 Arrays and Bits

(1) Für geordnete Mengen, wo jeder Eintrag einfach nur angibt ob das Element in der Menge ist oder nicht, können wir einfach die Bitoperationen anwenden um beispielsweise leicht die Anzahl an Elementen herausfinden die in beiden Mengen enthalten sind (durch den & Operator und danach rufen wir die count-Funktion auf). Zusätzlich können wir neue Mengen erstellen die...

- ...alle Elemente enthalten die mindestens in einer der beiden Listen enthalten sind.
- ...alle Elemente enthalten die nur in genau einer der beiden Listen enthalten sind.

Auch können wir sehr leicht das Komplement, Vereinigung, Durchschnitt und Differenz bilden. Natürlich geht noch viel mehr. Bitoperationen haben den Vorteil, dass sie deutlich schneller sind, als über einen Array zu iterieren.

(2) Negative vorzeichenbehaftete Integer haben ein Vorzeichenbit, dass auf 1 gesetzt ist. Ein logischer Rechts shift würde jetzt einfach alle bits nach rechts schieben und in die höchste Stelle eine 1. Somit wäre das Resultat dann plötzlich positiv.

Ein arithmetischer Rechtsshift lässt das erste Bit auf 1 und ersetzt die Bits nach dem Vorzeichenbit mit 1 (so viele wie geschoben wird). Damit bleibt die Zahl negativ und die Operation entspricht (etwa) einer Division mit 2, was unserer Vorstellung von einem Rechtsshift entspricht.

(3) $(n\&(n-1)) == 0$ testet ob die bitdarstellung von n und $n-1$ 0 ergeben. Das ist genau dann der Fall, wenn $n = 0, 1$ oder n eine 2er Potenz ist.Das ist der Fall, falls $n = 0$, $n = 1$ ist (&0 ergibt immer 0). Sei n also im Folgenden größer als 1. Angenommen n ist eine 2er Potenz (also an k.ter Stelle eine 1 im bitstring). Dann ist in n-1 genau jede 0 die vor der k.ten Stelle kommt auf 1 gesetzt und die 1 in der k.ten Stelle wird zu 0. In den höheren Stellen ändert sich nichts. Somit ergibt für diesen Fall die Formel auch 0.

Angenommen n ist nicht durch 2 teilbar. In diesem Fall werden in n-1 alle Nullen hinter der ersten 1 in n auf 1 gesetzt und die erste 1 auf 0. Nach der Annahme, dass n keine 2er Potenz ist, existiert jetzt ein höheres bit, dass sowohl in n als auch in n-1 auf 1 gesetzt ist. Somit ergibt die Formel nicht 0.

(4) Wir wollen die Anzahl der Einserbits in der Variablen sum speichern.

Wir gehen in einem for loop von 0 bis sizeof(integer) über jeden einzelnen bit indem wir die gegebene Zahl n immer um 1 nach rechts shiften und dann $n\&1$ auf unsere Summe addieren. $n\&1$ ist genau gleich dem letzten bit und so zählt dieser Algorithmus wie gefordert die Anzahl an gesetzten Bits.

Da es sich hier um einen unsigned Integer handelt sind keine weiteren Besonderheiten zu berücksichtigen.

(5) Gegeben sei ein unsigned Integer. Um die Bits nun zu reversen gehen wir mittels einer for Schleife über jedes einzelne Bit drüber. Das letzte Bit bekommen wir durch eine und-Verknüpfung mit einer Maske die Wert 1 besitzt. Jetzt wird der Integer in jedem Schritt um eins nach rechts geschoben und so bekommen wir der Reihe nach alle Bits. Diese werden nach einer Multiplikation auf einen Ergebnis Integer addiert (Ist mit 0 initialisiert). Der Wert des ersten Bits, was wir erhalten, wird mit 2^{31} multipliziert und auf das Ergebnis addiert (Ist mit 0 initialisiert). Das Nächste mit 2^{30} multipliziert usw. bis 2^1 und 2^0.

(6) Die Quicksortpartitionierungsfunktion gibt für eine übergebene Liste, eine Liste aus wo das Pivotelement bereits an der richtigen Stelle steht. Nun gehen wir die Liste durch und testen für jedes Element ob es größer als das Pivotelement ist. Ist das der Fall so tauschen wir es nach hinten.

Wenn wir Einträge nach gerade und ungerade sortieren wollen können wir ähnlich vorgehen nur, dass wir wir testen ob das Element gerade ist. Ist das der Fall so tauschen wir es nach hinten.

3 Arrays and Strings

(1) Als erstes testen wir ob das erste Zeichen ein $'-'$ ist, ist das der Fall so starten wir die for-Schleife Index 1 ansonsten bei Index 0. Wir initialisieren einen unsigned Integer mit 0 und starten die besagte for-Schleife. Diese geht von einem Index (entweder 0 oder 1) bis zur Länge des Strings alle Charakters durch. Für jeden Durchgang multiplizieren wir *result* mit 10 um alle bisherigen Zahlen um 1 nach links zu verschieben. Jeder Charakter wird der Reihe nach mittels $s[i] -' 0'$ zu einem Integer zwischen 0 und 9 konvertiert und auf *result* addiert.

(2) Um auf die einzelnen Dezimalziffern unseres Integers zuzugreifen verwenden wir den modulo Operator. Solange die gegebene Nummer nicht 0 ist rechen wir modulo 10, bekommen so die letzte Ziffer, wandeln diese mit $char(.) +' 0'$ in einen char um, der in Ascii genau der Ziffer entspricht und fügen diesen an unseren String an. Anschließend teilen wir die Nummer durch 10. Jetzt haben wir allerdings immer nur die kleinste Ziffer genommen also müssen wir am Ende noch den string umdrehen (reverse).

(3) ASCII benutzt 128 Zeichen, was relativ wenig sind, aber für viele Texte reicht es und somit ist die Kodierung optimal, weil wir pro Buchstabe nur 8 bit brauchen (bzw 7 aber das wird meist dann auch in 8 gespeichert).

EXTENDED ASCII ist ähnlich wie ascii nur haben wir das Doppelte an Bits (256) zur Verfügung, also brauchen wir zur Speicherung eines Charakters genau 8 bits.

UTF-8 ist eine dynamische Kodierung, d.h. wir haben einen Präfix der uns angibt wie viele bytes wir für einen Charakter lesen müssen. Damit können wir sowohl normales Ascii umsetzen als auch Sonderzeichen in mehrere Bytes kodieren. Es wird eine Länge von bis zu 4 Bytes unterstützt.

UTF-16 nutzt 16 oder 32 bits und hat somit auch dynamische Länge. Für eine normales Ascii Kodierung bietet sich dieses Format nicht an, weil wir dann immer 16 bits hätten wovon wir aber nur 7 benötigen.

UTF-32 nutzt 32 bit zur Speicherung und ist somit von fester Länge. Hier brauchen wir keinen Präfix, weil wir feste Länge haben, also nutzen wir unseren Platz besser aus. Andererseits wäre diese Kodierung eine Platzverschwendung für Texte mit Ascii Kodierung.

(4) Um einen String so schnell wie möglich zu säubern sollten vor allem inplace Replacement Verfahren verwendet und am Ende die redundanten Lücken die hierbei entstehen gelöscht werden. Falls Regex Ausdrücke benutzt werden versuche diese für mehr Geschwindigkeit und Effizienz hard zu coden. Nutze so wenig Fallunterscheidungen wie möglich.

(5) Die regex Klasse macht viele Fallunterscheidungen und ist somit relativ langsam.

(6) Run-length Encoding wird für Daten genommen, die viele sich wiederholende Charakters haben. Bei normalen Strings ist das meistens nicht der Fall also eher nicht. Eher dort wo immer ähnliche Muster auftreten oder es Blöcke, die nur einen Buchstaben haben gibt.

4 LinkedLists and Trees

(1) Sei eine verlinkte Liste und ein Element daraus gegeben, dass wir in konstanter Zeit löschen wollen.

Das geht indem wir die Daten aus dem Nachfolger in unseren Knoten schreiben und den pointer auf die übernächste Node (node->next->next) zeigen lassen. Das funktioniert nur wenn unser Knoten nicht das Ende der Liste ist. Ist das der Fall so setzen wir den Knoten einfach auf NULL.

(2) Sei erneut eine einzeln verlinkte Liste gegeben. Um jetzt einen Pointer zu erstellen der beispielsweise genau auf die Mitte zeigt können wir folgendermaßen vorgehen:

Wir erstellen 2 Pointer. Der eine läuft immer ganz normal einen Schritt nach vorne, während der Zweite immer 2 Schritte macht. Das Ganze kapseln wir in eine while Schleife, die läuft solange der schnelle Pointer nicht am Ende ist. Erreicht der schnelle Pointer jetzt das Ende so ist der langsame Pointer in der Mitte. Das geht deutlich schneller und effektiver als die naive Methode erst mit einem Pointer die Länge bestimmen und dann nochmal von vorne bis zur Mitte laufen.

(3) Statt eine extra Stack klasse zu implementieren benutzen wir einfach Vektoren und die dafür definierten Methoden $push_back()$, $empty()$, pop_back und $back()$.

(4) Um eine Liste als queue zu verwenden benutzen wir die Methoden $emplace\_back(.)$, die ein Element hinten an die Liste anhängt. Zusätzlich benutzen wir $front()$ um das erste Element zu erhalten sowie $pop\_front()$ um das erste Element zu entfernen.

(5) Ein BinaryTree ist einfach ein Baum, indem jeder Knoten maximal 2 Kinder hat. Ein BinarySearchTree hingegen ist ein BinaryTree, bei dem für jeden Knoten gilt: Das Maximum des linken Teilbaumes ist kleiner oder gleich als die Wurzel des Teilbaumes. Das Minimum des rechten Teilbaumes ist größer als die Wurzel des Teilbaumes.

(6) Es gibt 3 verschiedene Möglichkeiten einen BinaryTree zu traversieren:

- Preorder: Hier betrachten wir zuerst den jeweiligen Knoten wo wir sind, gehen dann erst nach Links und machen hier wieder das Selbe und anschließend noch nach rechts. Da wir erst die Knoten immer betrachten kann man mit diesem Algorithmus gut eine Kopie des Baumes machen.

- Inorder: Wir gehen erst nach Links, wenn im linken Teilbaum alles abgearbeitet ist betrachten wir unseren Knoten und gehen anschließend noch in den rechten Teilbaum und wiederholen das Ganze. Im Falle eines BinarySearchTree gibt uns dieser Algorithmus eine sortierte Liste mit allen Werten.
- Postorder: Bei postorder gehen wir erst von einem Knoten erst in den linken Teilbaum, dann in den Rechten und schließlich betrachten wir die Wurzel des Teilbaumes. Diese Reihenfolge hat zur Folge, dass wir erst alles Kinder eines Knotens betrachten, bevor der Knoten selbst an der Reihe ist. Somit eignet sich dieser Algorithmus zum Löschen eines Teilbaumes.

(7) Ein trie ist im Prinzip ein Baum wo jeder Knoten beliebig viele Kinder haben kann. Die Wurzel ist natürlich ausgezeichnet. In jedem Knoten steht ein Charakter und somit repräsentiert jeder Pfad von der Wurzel zum Blatt ein Wort.

Ein radix Tree ist ähnlich aufgebaut, nur sind an den Knoten statt einzelner Buchstaben, Strings gegeben (Können auch einzelne Buchstaben sein). Somit haben wir einfach nur einen Speicher effizienteren trie (unter Umständen kann es aber auch praktischer sein stattdessen einen Trie zu verwenden).

5 Heaps, Sorting und Hash Tabellen

(1) Ein Heap ist ein Binärer Baum, der vollständig ist (shape property). Dadurch kann man ihn als Array speichern. Zusätzlich gilt die heap property, also, dass der Wert im Elternknoten mindestens so groß ist wie in den Kindern (Max Heap).

(2) Wir benutzen einen Min Heap. Hierfür gehen wir den Daten Stream Elementweise durch. beinhaltet der Heap jetzt schon k Elemente und ist unser Wort kürzer als die Wurzel, so kommt das Wort nicht in Frage und wir gehen weiter. Ansonsten pushen wir das Element in die Wurzel und falls die Länge jetzt größer als k ist, so löschen wir die Wurzel (ist das kürzeste Element nach der Definition von min Heap). Am Ende lesen wir des Heap aus.

(3) Auch hierfür benutzen wir einen Min Heap. Wir beginnen damit für jeden Datenvektor einen Verweis auf die Liste in den Min Heap zu pushen. Ist der Min Heap fertig aufgebaut, so popen wir das top Element (Kleinstes nach Definition von min Heap). Aber das Element ist ja nur ein Verweis auf eine längere Liste. Insofern schauen wir jetzt noch ob danach noch Elemente kommen. Ist das der Fall so pushen wir den Verweis auf das nächste Element wieder in den Min Heap. Das machen wir solange bis der min Heap leer ist.

(4) Sorting Networks sind aus kleinen Bauelementen (die im Prinzip 2 gegebene Zahlen entweder vertauschen oder nicht) aufgebaut und mit ihnen kann man Sortieren hart codieren. Für kleine Array Länge ist das schneller als normale Sortieralgorithmen.

(5) Sortierte Arrays sind praktischer, weil man nicht immer den ganzen Array durchsuchen muss um ein Element zu finden. In einem sortierten Array steht dieses Element an einer ganz bestimmten Stelle. Für viele Mengenoperationen (bspw. Schnitt oder Vereinigung) muss man ständig Elemente suchen und mit sortierten Arrays braucht man jetzt nur die Arrays Elementweise durchgehen und vergleichen.

(6) Gegeben sei eine Hash Funktion die eine Menge von Wörtern auf eine Hash Tabelle abbildet. Hat diese Funktion keine Kollisionen so ist es eine perfekte Hash Funktion. Hat der Hash zusätzlich in seiner Hash Tabelle keinen ungenutzten Platz, so heißt er minimal. Ein minimal perfekter Hash ist somit ein Hash der keine Kollisionen bildet und jeden Platz in der Hashtabelle ausnutzt.

6 Algorithm Design

(1) Backtracking ist ein Verfahren, das ein Problem löst indem es einzelne mögliche Züge simuliert und, falls es einen Widerspruch gibt, zurücksetzt. Für höhere Effektivität benötigen wir ein gutes prunning Verfahren.

(2) Wir beginnen damit eine Queen in die erste Reihe zu platzieren (zufällig, aber mit gewissem System, sodass alle möglichen Plätze irgendwann benutzt wurden). Jetzt schauen wir wo eine Queen in die nächste Reihe gesetzt werden kann (ebenfalls zufällig, aber mit System). Das machen wir solange bis jede Reihe eine Queen hat oder ein Fehler auftritt (in diesem Fall setzen wir das Brett um einen Zug zurück). Haben wir eine vollständige Lösung erhöhen wir einen vorher mit 0 initialisierten Zähler um eins und setzen den letzten Zug zurück. Das machen wir solange bis es keine möglichen Lösungen mehr gibt.

(3) Um das 15 Puzzle zu lösen müssen wir auf jeden Fall eine Liste mit allen bisher besuchten States mitführen, um keine Loops zu kreieren (Das Board schicken wir in eine Hash Funktion um effektiver zu speichern.).

Wenn wir jetzt alle Kandidaten für den nächsten Zug suchen schauen wir für alle möglichen Züge ob der Board State schon erreicht wurde (Kollision in der Hashtabelle). Ist das der Fall so ignorieren wir diesen Kandidaten. Wenn nicht, so hängen wir den jetzigen Hash an die Tabelle an und hängen den Kandidaten zudem an den Lösungsvektor an. Der Lösungsvektor beinhaltet alle Schritte die zu dieser Lösung geführt haben. Kommen wir in eine Sackgasse so löschen wir das letzte Element und suchen für das Element davor einen anderen Folgezustand.

Um effektiv zu prunen ist es außerdem hilfreich eine Abstandsfunktion einzuführen, die abschätzt wie viele Züge wir mindestens noch brauchen um auf die perfekte Lösung zu kommen.

(4) Meist führt man schon sehr viel aus indem man das Problem in Subprobleme zerlegt. Auf der ersten Ebene ist eine Parallelisierung nicht trivial, da wir ja das ganze Problem erst aufteilen müssen.

Beispielsweise müssen wir für quicksort erst die gesamte Liste nach dem Pivot teilen. Hier ist eine Parallelisierung nicht ganz trivial. Danach können wir einfach für jede entstandene Hälfte einen Thread starten.

(5) Divide and Conquer sollte, wie der Name es sagt, das Problem in kleinere Probleme aufteilen. Im Falle von beispielsweise recursive binary Search betrachten wir zu Beginn das Ganze Problem, danach allerdings nur noch einen Teil (beispielsweise Hälfte) des Teilproblems, die anderen Teile interessieren uns nicht mehr. Also verkleinern wir das Problem eher (decrease), als es zu teilen.

(6) Dynamische Programmierung funktioniert ähnlich wie Divide and Conquer nur das wir hier bei der Lösung für ein Teilproblem suchen ob wir das Problem schon einmal gelöst haben (in beispielsweise einem Vektor). Ist das der Fall so schreiben wir die Lösung einfach ab. Falls nicht so lösen wir das Problem und hängen die Lösung an de besagten Vektor an. Somit verhindern wir, dass wir Dinge doppelt berechnen.

(7) Dynamisches Programmierung bottom up startet von allen Teilproblemen und löst die darüberliegenden Probleme mit diesen. Das ist meist leichter zu verstehen und wir brauchen für die Lösungen nur einen kleineren Cache (haben wir die unterste Ebene gelöst so brauchen wir die Werte meist nicht mehr, beispielsweise für die Fibonacci Zahlen brauchen wir 0 und 1 für die erste Ebene, aber danach nur noch 1, 2 bei richtiger Implementierung, also können wir 0,1 aus dem Cache löschen).

Bei top down starten wir mit der Aufgabe und zerlegen diese meist rekursiv in Teilprobleme. Diese lösen wir und speichern sie in einem Cache. Hier haben wir den großen Vorteil, dass wir pruning anwenden können, also das Abschneiden von Ästen die nicht mehr zu einer optimalen Lösung führen können.

(8) Angenommen wir haben Gegenstände mit unterschiedlichen Werten und Gewichten. Wir wollen so viel Wert wie möglich wegtragen, aber können nur maximal ein bestimmtes Gewicht tragen. Ein greedy Algorithmus könnte jetzt beispielsweise sagen, dass wir, solange wir noch mehr tragen können, den wertvollsten Gegenstand, der leicht genug ist, einpacken. Das führt in manchen Szenarios aber nicht zur optimalen Lösung. Wir können das jetzt verändern indem wir das Verhältnis von Wert zu Gewicht betrachten.

7 Compilers

(1) Um einen Syntax Baum aus einer Grammatik zu erstellen müssen wir zuerst einen Scanner schreiben, der den Input in Symbole verwandelt. Danach bauen wir einen Parser, der für die gegebene Grammatik kontrolliert ob die Sequenz von Symbolen, die uns unser Scanner gibt, den Regeln der Grammatik entspricht.

Daraus bauen wir jetzt einen Syntax Tree, indem wir, falls sich eine Operation auf 2 oder mehr Operanden bezieht, diese Operation in die Wurzel schreiben und die

Operanden in deren Blätter. Falls wir nur einen Operanden haben schreiben wir diesen einfach in das linke Kind des Operationsknoten. Somit bauen wir sukzessiv einen AST auf. Schlussendlich können wir den Syntax Tree noch optimieren.

(2) Der Scanner liest den gegebenen Input lediglich und ordnet Zeichenketten daraus ihre "Klasse" (Symbole) zu. Jede Zeichenkette eines Types bekommt einen Identifizierer (Wert) und eine Beschreibung (Klasse).

(3) Für die Regel

$$d = \{'a'\}['b']'c'$$

ist der Parser folgendermaßen aufgebaut.

Am Anfang können beliebig viele "a"s vorkommen. Also checken wir ob wir ein "a" finden, und wenn das der Fall ist dann lesen wir mittels einer while Schleife solange bis wir einen anderen Buchstaben finden. Wenn nicht gehen wir sofort zum nächsten Schritt weiter.

Danach schauen wir ob wir ein "b" finden. Entweder wir haben genau ein "b" oder keins. Anschließend kontrollieren wir, dass der nächste Buchstabe ein "c" ist und dann nichts mehr kommt. Ist eine der Bedingungen verletzt so werfen wir einen Fehler.

(4) Wir können Teilbäume vereinfachen. Zum Beispiel falls eine Operation in der Wurzel steht können wir diese auf ihre Kinder anwenden und die Operation mit dem Ergebnis ersetzen.

Weiterhin können wir gleiche Teilbäume mit einer Variablen kodieren und durch diese ersetzen.

Außerdem können wir für komplexere Probleme Verzweigungen vorhersagen und gegebenenfalls eliminieren. Falls wir beispielsweise eine Multiplikation haben und in unserem ersten Knoten eine 0 steht, so wissen wir sofort, dass das Ergebnis auch 0 ist und können das direkt in den Elternknoten schreiben ohne die anderen Knoten zu betrachten.

8 Programming Competition

(1) Die nächste Aktion wird mit einem deterministischen endlichen Automaten bestimmt: Es wird zuerst getestet ob wir ein Gericht abgeben können. Ansonsten schauen wir nach ob alle Zutaten die benötigt werden existieren. Ist das der Fall so werden sie einfach eingesammelt. Falls das nicht der Fall ist, wird zuerst gecheckt ob der Ofen gerettet werden muss (Croissant oder Tart im Ofen), wenn nicht so stellen wir die fehlende Zutaten her.

(2) Im Verlauf des Programmierprozesses kam es immer wieder zu suboptimalen Verhalten des Bots. Das lag meist daran, dass bestimmte Zwischenfälle nicht beachtet bzw. nicht abgefangen wurden.

Außerdem kam es immer wieder zu Problemen, weil der andere Spieler irgendeinen Quatsch gemacht hat (Teller nicht fertig gestellt und einfach abgelegt, im Weg stehen, etc.). Durch eine sinnvolle Fallunterscheidung kann das meist aber ebenfalls umgangen werden.

Des Weiteren wird in unserem Modell immer eine feste Reihenfolge zum Einsammeln der Zutaten eingehalten. Das kann zu sehr ineffizienten Wegen führen.

(3) Eine Heuristik die implementiert wurde war, dass alle Zutaten nicht der Reihe nach durchgegangen werden. Stattdessen schauen wir was relativ zum Charakter die Nächstgelegene ist. Beispielsweise, wenn wir Blaubeerren und Eis benötigen gehen wir zuerst zum Eis, da es näher an unserem Charakter liegt.

Zu Beginn einer Runde kontrollieren wir jeden herumliegenden Teller, ob er für unser Gericht benutzt werden kann. Das verhindert herumliegende Teller sowie fast fertige Gerichte, die nie abgegeben werden.

Um Dinge abzulegen benutzen wir nicht das Offensichtliche: Einfach alle Tiles durchgehen und am Erst gefundenen leeren Tile ablegen, sondern wir suchen das nächstgelegene Tile zu unserem Spielcharakter und legen dort ab.

Da Tart und Croissant am Schwersten zum Zubereiten sind, sorgen wir dafür, dass der Ofen so gut wie immer arbeitet. Sollten wir in irgendeinem Gericht ein Croissant brauchen und existiert auf dem Spielfeld gerade keins, so schmeißen wir einen Teig in den Ofen. Das Gleiche machen wir für den Kuchen. Das war eine enorme Verbesserung für den Bot, da es, falls der andere unseren Kuchen genommen hat, sofort einen neuen Kuchen gab.

9 Graph Traversion

(1) Gesucht sei der kürzeste Pfad zwischen 2 Knoten in einem Graphen wo alle Kanten die Länge 1 haben. Da die Kanten alle mit 1 gewichtet sind führt eine simple Breitensuche zum Ziel. Diese hat, im Gegensatz zu Dijkstra und Floyd-Warshall, den Vorteil, dass es deutlich intuitiver funktioniert und die Laufzeit schneller ist.

(2) Um sehr schnell einen funktionierenden Algorithmus zu schreiben, nimmt man am Besten den Floyd Warshall Algorithmus, da dieser sehr schnell zu implementieren ist. Dijkstra zu implementieren dauert zwar nur etwas länger, aber sollten die zufälligen Kantengewichte zufällig negativ sein, so funktioniert dieser Algorithmus nicht. Da der Graph nach Aufgabenstellung zudem nicht so groß ist, können wir problemlos den Floyd-Warshall Algorithmus verwenden.

(3) Gegeben sei ein Graph mit negativer Kantenlänge (mindestens eine). Wendet man hier Dijkstras Algorithmus an, so wird unter Umständen eine nicht perfekte Lösung

gefunden. Das Problem ist, dass der Dijkstra Algorithmus abbricht, wenn der Weg bis zu einem neuen Knoten schon länger ist als der bereits gefundene Weg. Im Falle von negativen Kanten kann dieser längere Weg aber verkürzt werden und so gegebenenfalls das neue Minimum annehmen.

10 Minimum Spanning Trees

(1) Angenommen wir haben einen zyklischen Graphen gegeben. Also gibt es einen Pfad von Knoten x nach Knoten y und von diesem wieder zurück zu x. Topologisches Sortieren erzeugt nun eine Reihenfolge der Knoten wo v_i vor v_j genau dann, wenn es einen (gerichteten) Pfad von v_i nach v_j gibt. Somit müsste in unserem Beispiel in der Sortierung x vor y kommen. Da es aber gleichzeitig (nach Voraussetzung) einen Weg zwischen y und x gibt müsste y vor x stehen. Das ist ein Widerspruch.

(2) Beim Eulerproblem 1 ist ein Algorithmus gesucht, der die Summe aller durch 3 oder 5 teilbaren Zahlen zwischen 1 und 1000 (ausschließlich) findet. Das geht einfach mit einer for Schleife über alle Zahlen zwischen 1 und 1000 (ausschließlich). Für jede Instanz des for-loops machen wir eine if Abfrage ob die Zahl durch 3 oder 5 teilbar ist (einfach Modulo rechnen). Ist das der Fall so addieren wir die Zahl.

(3) Das zweite Eulerproblem ist etwas komplexer: Hier ist die Summe der Kantenlängen des minimal Spanning trees gesucht (bzw. die Einsparungen zum originalen Graphen). Dazu bauen wir schrittweise einen minimum Spanning tree mit Hilfe von Prim's Algorithmus auf. Ich habe Prim benutzt, weil man für Kruskal's Algorithmus zusätzlich noch nach Kreisfreiheit kontrollieren muss. Da kein minimaler Spannungsbaum sondern nur die Summe der Kanten gefordert war ist meine Version etwas vereinfacht, macht aber im Grunde genau das Selbe. Bereits beim Einlesen der Adjazensmatrix wird die Summe der Kanten im originalen Graphen berechnet.

11 Maximum Flow

(1) Der Algorithmus von Edmonds und Karp wird benutzt um den maximal möglichen Fluss zwischen 2 gegebenen Knoten innerhalb eines Graphen zu berechnen. Die Idee ist es erst einmal einen Pfad zwischen dem Start -und Endknoten zu finden und für diesen den Fluss berechnen. Nun geht man mittels einer Breitensuche alle möglichen Pfade durch und updatet den Fluss.
Hier hat man noch das Problem, dass die Lösung nicht optimal sein muss (hängt von der Reihenfolge der Bearbeitung ab). Das beheben wir mit der Einführung von Rückwärtskanten. Wenn ein gerichteter Pfad zwischen 2 Knoten in eine Richtung genommen wird und somit der maximal mögliche Fluss in dieser Kante reduziert wird, wird gleichzeitig eine entgegengesetzte Kante eingeführt, deren maximal mög-

licher Fluss um den gleichen Wert erhöht wird, wie die andere Kante verringert. Das wirkt erst einmal gegenintuitiv, aber führt zu einer optimalen Lösung.

(2) Gesucht ist die maximale Anzahl an Iterationen die der Algorithmus für eine Zahl zwischen 2 gegebenen Werten benötigt ($3n + 1$ Algorithmus).

Für eine gegebene Zahl wird der Algorithmus rekursiv ausgeführt und gibt die Anzahl der Iterationen zurück. Diese wird anschließend mit dem Maximum verglichen und ersetzt dieses gegebenenfalls. Mit einer for Schleife wird über alle Zahlen in der gegebenen Range iteriert.
Der rekursive Algorithmus benutzt ein paar "Abkürzungen".
Es fällt auf, dass er in $log2(zahl)$ terminiert, wenn $zahl$ eine 2er Potenz ist (prunning). Das ist möglich, da hierfür die Anzahl der folgenden Schritte fest steht.

Außerdem wird sehr oft für eine Zahl die selben Iterationen ausgeführt. Das kann mit einer Map verhindert werden. In dieser werden alle berechneten Werte gespeichert und können jeder Zeit ausgelesen werden. Der Nachteil ist natürlich, dass wir suchen müssen und außerdem mehr Speicherbedarf haben.

(3) Gesucht ist die Anzahl der Zusammenhangskomponenten für einen Graphen mit gegebener Knotenanzahl und Adjazensmatrix (beide Informationen sind leicht aus dem Input auszulesen).

Zur Lösung dieser Aufgabe benutzen wir im Prinzip Dijkstra's Algorithmus (mit Kantengewichten 1). Dieser bestimmt, im Gegensatz zu Floyd Warshall, für einen gegebenen Startknoten den kürzesten Abstand aller anderen Knoten im Graph. Floyd Warshall bestimmt paarweise die kürzesten Abstände, aber eigentlich interessieren uns nur die Zusammenhangskomponenten, also ob es einen Weg gibt. Dijkstra hat eine Laufzeit von $O(|V| + |E| \log |V|)$ und Warshall eine Laufzeit von $O(|V|^3)$. Da Dijkstra für unser Problem ausreicht wählen wir den Algorithmus auf Grund seiner deutlich besseren Laufzeit. Nach einer Iteration müssen wir lediglich checken ob es noch Knoten gibt, die nicht in der Knotenliste auftauchen (bzw. Abstand ∞). Für diese müssen wir Dijkstra nochmal ausführen.
Offensichtlich ist ist der Wert des kürzesten Weges für diese Aufgabe aber nicht relevant. Damit können wir den Algorithmus vereinfachen und nur eine Breitensuche pro Zusammenhangskomponente machen.

12 Minimum Cut

(1) Wir teilen die Aufgabe in 2 Teile: Im ersten erstellen wir einen Graphen und im Zweiten berechnen wir mit dessen Hilfe die Lösung.

Wir beginnen mit der Erstellung von Knoten s und t. Dann erstellen wir für jeden Arbeiter sowie jede Arbeit einen Knoten. Von jedem Arbeiter gibt es Kanten (mit Gewicht 1) zu einer Aufgabe genau dann, wenn der Arbeiter für diese Aufgabe qualifiziert ist. Von Startknoten s gibt es zu jedem Arbeiter eine Kante, die

mit der Kapazität des jeweiligen Arbeiters gewichtet ist. Von jeder Arbeit gibt es eine Kante nach Endknoten t, die mit dem Zeitaufwand für die jeweilige Arbeit gewichtet ist.

Nun wenden wir den Maximum Flow Algorithmus darauf an und finden eine optimale Verteilung der Arbeiten bzw. Arbeiter. Nun müssen wir lediglich den Arbeitern die Arbeiten zuweisen, die im Maximum Fluss Graphen eine Verbindung haben. Im Pseudocode sieht das dann folgendermaßen aus (Sei $w.capacity$ für Worker w, die Kapazität des Arbeiters):

```
// create graph
V = {s,t}
E = {}
for w in workers:
        V.add(w)
        E.add((s,w,w.capacity))
for a in tasks:
        V.add(a)
        E.add((a,t,a.cost))
        for w in workers:
                if (w.qualified(a)):
                        E.add((w,a,1))

// compute maximum flow and assign tasks to each worker
Flow = maximum_flow(V,E,s,t)
for w in workers:
        for edge in Flow.edges:
                w.assign(edge.end_node)
```

(2) Das MinCut-Problem sucht in einem Graphen eine Minimale Kanten-Teilmenge (minimale Kantengewichte), die 2 gegebene Knoten voneinander trennt.

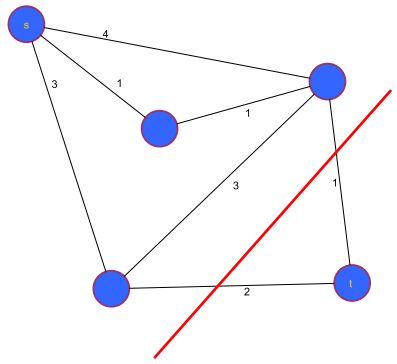

Abbildung 1: Genau ein Minimum Cut.

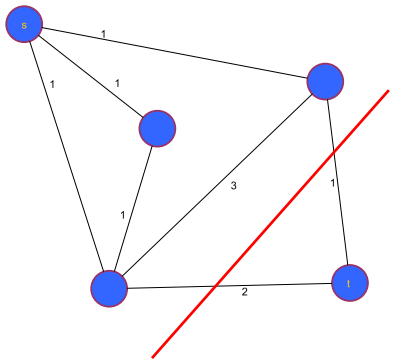

Abbildung 2: 3 Minimum Cuts.

(3) Beim Matching Problem sind (genau) 2 disjunkte Knotenmengen gegeben. Es kann Kanten zwischen diesen Knotenmengen geben, aber keine zwischen Knoten der selben Menge. Ein Matching ist jetzt eine Menge von Kanten, sodass jeder von dieser Kantenmenge erreichte Knoten nur einmal vorkommt.

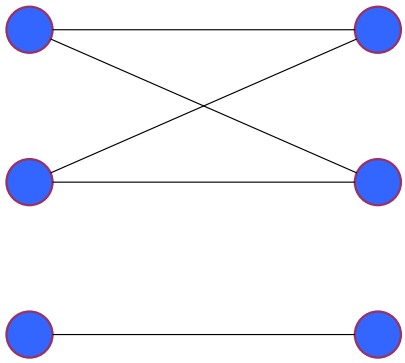

Abbildung 3: 2 perfekte Matchings.

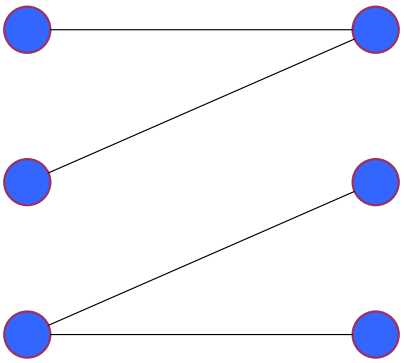

Abbildung 4: Kein perfektes Matching.

13 Code Golf

(1) Die sicherlich drastischste Änderung war sich mit 3 Input Variablen zufrieden zugegeben. Das ist möglich, da wir viele der Input Variablen überhaupt nicht verwenden. Inputs die nicht benötigt werden, können einfach in ein und die Selbe Variable geschrieben werden, weil wir den Wert ja nicht mehr benötigen. Außerdem können verwendete Variablen später wieder benutzt werden. Das hat uns auch etwa 60 Zeichen gespart.

(2) Hierfür würde ich den Floyd Warshall Algorithmus vorschlagen: Wir berechnen zwar sehr viel was wir später nicht mehr brauchen, aber im Code Golf geht es ja primär nicht um die Laufzeit sondern die Menge der verwendeten Zeichen. Floyd Warshall benutzt wesentlich weniger Zeichen als Dijkstra und bietet sich somit an.