dokumentation

36 Bundeswettbewerb Informatik  
2. Runde / Aufgabe 1 und 3

# Aufgabe 1: Die Kunst der Fuge

## Lösungsweg:

Nachdem ich mich zuerst mathematisch mit der Aufgabe auseinandergesetzt hatte, hatte ich recht schnell die Formeln, um weitere Eigenschaften der Mauer in Abhängigkeit von N (Anzahl der Klötzchen) herzuleiten. Danach habe ich sehr viele unterschiedliche Strategien ausprobiert, um eine Mauer mit N Klötzen direkt zu konstruieren. Jedoch hat kein Algorithmus für ein N größer als 7 funktioniert. Nach einiger Zeit habe ich im Internet nach ähnlichen Problemstellungen gesucht aber auch da nichts gefunden, was mir weiterhalf. Daraufhin wollte ich erstmal mit einem Programm alle möglichen Mauern der maximalen Höhe brute-forcen und ausgeben lassen, da ich hoffte in einer der ausgegebenen Mauer ein Muster zu erkennen, aus dem ich einen Algorithmus ableiten konnte. Allerdings war meine Brute-force Algorithmus für N = 10 bezüglich Rechenzeit und Speicherplatz trotz Optimierungen zu komplex und so entschloss ich mit letztendlich den Brute-force Algorithmus so abzuwandeln, dass er mir nicht alle möglichen fertigen Mauern ausgibt, sondern abbricht sobald er eine Mauer gefunden hat.

## Umsetzung:

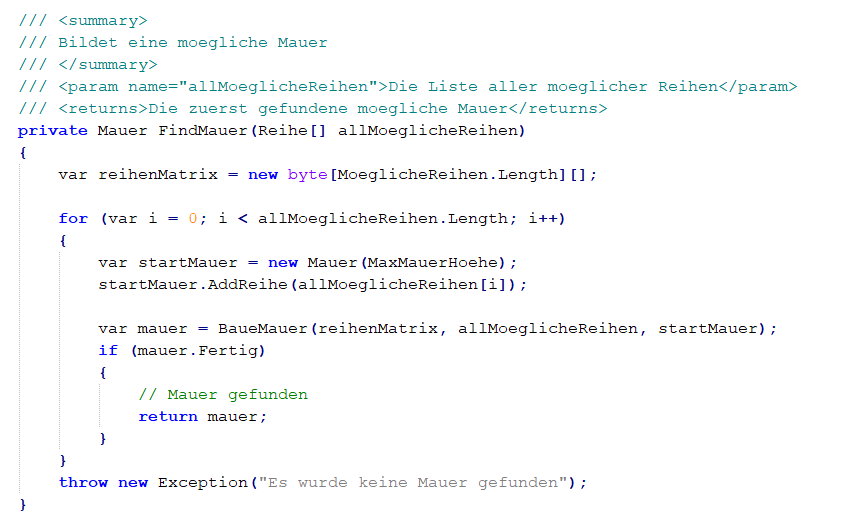
Das Programm ist eine Konsolenanwendung, geschrieben in C# mit Visual Studio. Zu Beginn wird der Anwender aufgefordert N zu definieren. Die Eingabe muss eine Zahl zwischen 2 und 22 sein. Bei 1 kann eine unendlich hohe Mauer gebildet werden, da eine Reihe mit nur einem Baustein keine Fugen besitzt und die Obergrenze von N = 22 ist durch die Wertebereiche mancher Datentypen bedingt. Kommt es zu einer ungültigen Eingabe, wird der Anwender erneut aufgefordert eine Zahl einzugeben.

Wurde eine gültige Zahl für die Anzahl der Klötzchen definiert, werden zunächst in der Methode BerechneEigenschaften der Klasse WallBuilder die Mauer Breite, die Anzahl der möglichen Fugenstellen, die maximal mögliche Mauerhöhe und die Anzahl der Fugen, die bei einer Mauer der maximalen Höhe benutzt werden müssen, berechnet.

Daraufhin wird die Methode StartAlgorithmus der Klasse WallBuilder ausgeführt. In dieser wird als erstes eine Liste aller Zahlen bis einschließlich N erstellt, die im nächsten Schritt für die Erstellung aller Permutationen verwendet wird. Die Erstellung der Permutationen erfolgt über eine optimierte Implementierung des Heaps Algorithmus. Die Methode SammlePermutationen der Klasse Utilities enthält als zweiten Parameter einen Delegaten der ein Byte Array als Übergabeparameter hat und einen Boolean zurückgibt. Somit kann nach jeder einzelnen Permutation über den Rückgabewert der referenzierten Methode entschieden werden, ob weitere Permutationen gesucht werden sollen. Im aktuellen Fall werden jedoch immer alle Permutationen gesammelt, sodass man am Ende ein Array aus Byte Arrays hat, wobei jedes Byte Array eine Permutation darstellt.

Für jede Permutation wird nun eine Reihe Instanz erstellt, die die Folge der Klötze als Byte Array speichert und in einem HashSet vom Typ Byte die Indizes der Fugen beinhaltet, die diese Reihe besetzt. Nachdem jede Reihe über das MoeglicheReihen Array referenziert wurde, beginnt der Mauerbau.

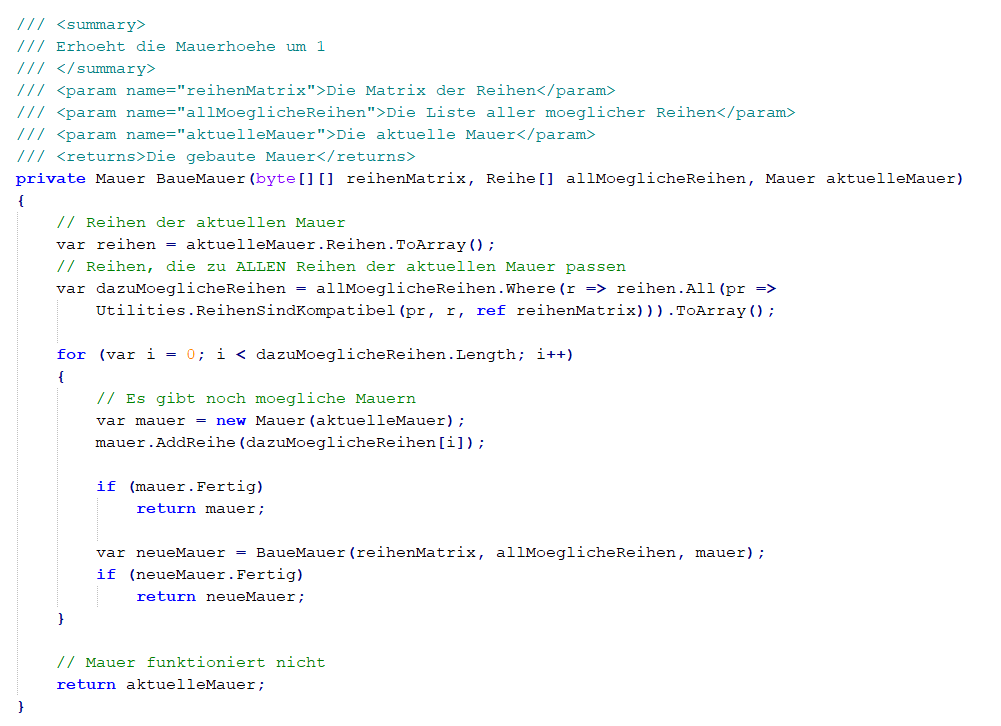
Zuerst wird die Methode FindMauer (siehe Abbildung 1) in der Klasse WallBuilder ausgeführt und ihr wird das Array aller Permutations-Reihen übergeben. Die Methode erstellt eine neue Variable reihenMatrix die ein verzweigtes Byte Array ist. In dieser Matrix wird eingetragen, ob zwei Reihen kompatibel sind (Keine Fugen überlappen sich). Ist dies der Fall hat das Feld der beiden Reihen den Wert 2. Sind die zwei Reihen nicht kompatibel bekommt das Feld den Wert 1. In die Matrix wird jedoch nur ein Wert eingetragen, wenn zwei Reihen auch verglichen werden.



Abbildung

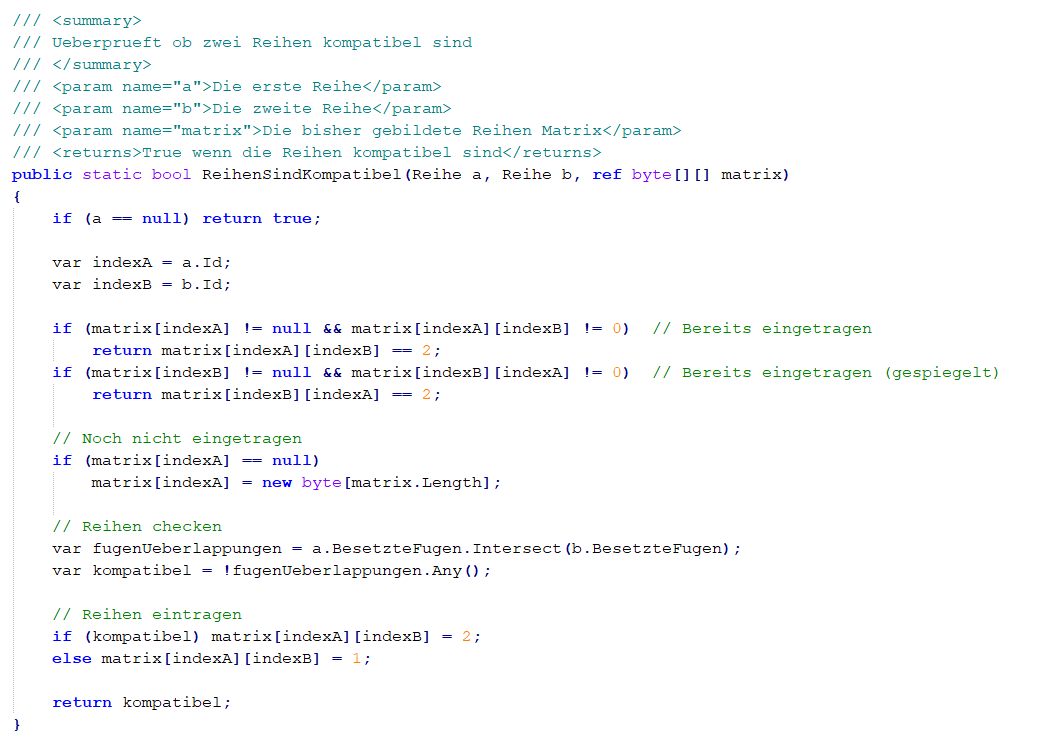
Der Brute-force Algorithmus beginnt indem er in einer Schleife alle möglichen Reihen durchgeht. Aus der jeweiligen Reihe wird dann eine Mauer startMauer erstellt. Die Klasse Mauer hat ein Array vom Typ Reihe, dass die Reihen der Mauer referenziert, ein HashSet vom Typ Byte, dass die Indizes der besetzten Fugen in der gesamten Mauer beinhaltet und einen Boolean Fertig der angibt, ob alle Reihen der Mauer gesetzt sind. Für die gebaute Mauer wird dann die Rekursive Methode BaueMauer aufgerufen.

Die Methode BaueMauer (siehe Abbildung 2) sammelt zuerst alle Reihen, die zur übergebenen Mauer passen und erstellt dann wiederrum für jede dazu passende Reihe eine neue Mauer Instanz mit den gleichen Reihen wie der übergebenen Mauer und fügt dieser Mauer die jeweilige passende Reihe hinzu. Ist die erstellte Mauer nun fertig, sprich alle ihre Reihen sind gesetzt, wurde eine Mauer der maximalen Höhe gefunden und diese Mauer kann zurückgegeben werden. Ansonsten ruft sich die Methode selber auf und übergibt die neu erstellte Mauer.



Abbildung

Die Methode ReihenSindKompatibel (siehe Abbildung 3) überprüft ob zwei Reihen kompatibel sind. Als erstes wird überprüft, ob es schon einen Eintrag für beide Reihen gibt. Gibt es noch keinen Eintrag wird überprüft ob beide Reihen eine Fuge mit dem gleichen Index in ihrem BesetzteFugen HashSet. Ist dies der Fall sind die Reihen nicht kompatibel.



Abbildung

Zum Schluss wird die gefundene Reihe noch in der Konsole ausgegeben.

## Beispiele:

|  |  |
| --- | --- |
| N = 6  Laufzeit: Ø 30ms  Verbrauchter Speicherplatz: Ø 10MB |  |
| N = 8  Laufzeit: Ø 64.000ms  Verbrauchter Speicherplatz: Ø 70MB |  |
| N = 9  Laufzeit: Ø 10.000ms  Verbrauchter Speicherplatz: Ø 165MB |  |
| N = 10  Laufzeit: ?  Verbrauchter Speicherplatz: Ø 1,5GB |  |

# Aufgabe 3: Quo vadis, Quax?

## Lösungsweg:

Um die möglichst wenige Gebiete zu scannen überlegte ich mir, dass die Drohne zuerst ein möglichst großes Gebiet scannen muss und dann dieses Gebiet, wenn es gemischt ist, in kleinere Gebiete zerlegen muss, die dann wiederrum gescannt werden. Das heißt im einfachsten Fall, einer quadratischen Map ohne Wasser (siehe Abbildung 4), bräuchte es nur einen Flug, um den Weg von Quax zur Stadt zu finden.

Wenn ein gescanntes Gebiet gemischt ist, ist es am effizientesten das Gebiet in 4 gleichgroße Teilgebiete aufzuteilen (siehe Abbildung 5).

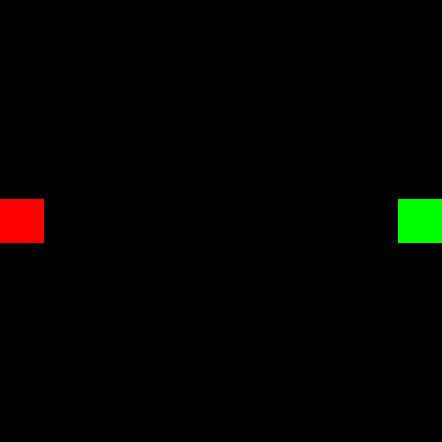
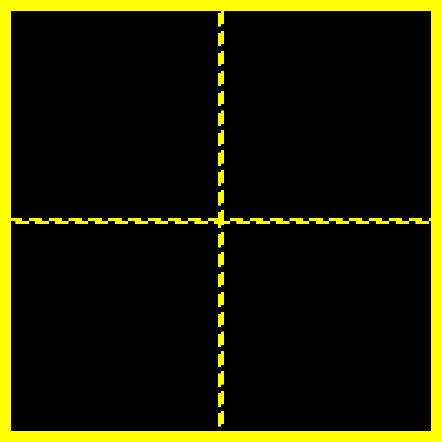


Abbildung 5

Abbildung 4

Die daraus resultierende Datenstruktur (siehe Abbildung 6) heißt Quadtree oder auch Quaternärbaum.

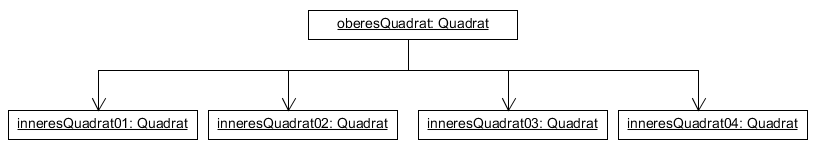


Abbildung 6

Nun wollte ich diese Datenstruktur für meinen Algorithmus verwenden. Allerdings war der Start anders als bei anderen Pathfinding Algorithmen. Während andere Algorithmen schon beim Ausgangspunkt, also hier die Position von Quax, anfangen, fängt mein Algorithmus erst beim größtmöglichen Quadrat an. Das heißt also, dass mein Quadtree als erstes Quax suchen muss. Das geht noch relativ einfach, indem man einfach so lange das Quadrat indem Quax sich befindet in innere Quadrate aufteilt, bis das Quadrat das Quax enthält Land ist, oder gemischt und eine Größe von 20x20 hat.

Hat man das Quadrat indem Quax ist herausgefunden, ist der nächste Schritt die anderen inneren Quadrate des oberen Quadrats des Quax-Quadrats zu untersuchen. Und sortiert nach der Entfernung zur Stadt.