Aufgabe 2: Alles Käse

Teilnahme-ID: 66048

Bearbeiter/-in dieser Aufgabe:
Karolina Schnitter

11. February 2023

Inhaltsverzeichnis

Lösungsidee....1
Umsetzung....1
Beispiele....2
Quellcode....2

Lösungsidee

Ich habe die Aufgabe in 2 separate Lösungsschritte unterteilt:

1. Käsescheiben finden, die praktisch der “Kern” (die letzten Scheiben, die nach dem Schneiden des Käses übrig bleiben) des Käsequaders waren

2. alle restlichen Käsescheiben auf diesen “Kern” legen

Zu 1.

Um die Anzahl der möglichen Start-Käsescheiben einzugrenzen habe ich folgenden Ansatz verwendet:

Damit eine Käsescheibe eine Start-Käsescheibe sein kann, muss sie mehrmals vorhanden sein zb die Käsescheibe [x, y] ist c mal vorhanden, dann kann der Startquader die Längen [x, y, z] mit z <= c haben. Wenn es eine andere Käsescheibe [x, z] oder [y, z] gibt, kann diese an den Startquader gelegt werden. Wenn es eine andere Käsescheibe [c, d] mit d < x und d < y gibt, kann diese niemals an den Startquader angelegt werden. Es ist also sinnvoll nach einer Käsescheibe [x, z] oder [y, z] zu suchen, für die z möglichst klein ist und dann für alle möglichen Werte von x und y zu schauen, ob es Käsescheiben mit einem dieser Werte und der Anzahl c oder mehr gibt. Diese Käsescheiben sind dann mögliche Kandidaten für den Startquader.

Zu 2.

Dieser Schritt lässt sich mit Backtracking lösen. Die Längen des Quaders werden getrackt. Jedes mal, wenn eine neue Scheibe angelegt werden soll, wird in einer effizienten Datenstruktur (z.B. Hash Set oder Hash Map) nachgeschaut, welche der Scheiben [x, y], [x, z] oder [y, z] vorhanden ist.

Dann werden die Längen und die Menge der Scheiben aktualisiert. Wenn keine passende Scheibe in der Menge vorhanden ist, wird die letzte Scheibe entfernt und von da aus nach einer neuen Scheibe gesucht. Um die Reihenfolge der Scheiben zu speichern, eignet sich ein Stack, da jeweils nur eine Scheibe hinzugefügt oder abgenommen werden muss.

Laufzeit:

Der Aufbau des Käsewürfels kann als Baum mit maximalem Verzweigungsgrad von 3 und Höhe n dargestellt werden. Der oben beschriebene Algorithmus ist praktisch depth-first search auf diesem Baum.

1. Der Baum ist vollständig, wenn man annimmt, das eine Scheibe potentiell zu jeder Zeit an jeder Seite angelegt werden kann. Im worst-case müssten alle Blätter überprüft werden, was eine Zeitkomplexität von O(3^n) bedeutet.

2. Sehr oft hat die worst-case Zeitkomplexität keinen Einfluss auf die tatsächliche Laufzeit. Bei den Beispieleingaben lassen sich die Laufzeiten nur bei den drei größten Eingaben (siehe Tabelle) vergleichen, da bei den kleineren Eingaben der eigentliche Algorithmus keinen großen Teil der Zeit in Anspruch nimmt. Eingabe und Ausgabe sind immer O(n). Die Zeitkomplexität liegt also zwischen O(n) und O(3^n)

	Test 5	Test 6	Test 7
Zeit (in nanosecs) / n	251	275	318
n	6848	90300	1529903

Aufgabenteil B:

Als Erweiterung von Aufgabe A habe ich gewählt, dass mehrere Scheiben zwischendurch aufgegessen werden können. Diese können kein Teil des Kerns des Quaders sein.

Der Algorithmus wird erweitert: Beim Aussuchen der nächsten Scheibe wird zuerst überprüft, ob eine passende Scheibe in der Menge vorhanden ist. Wenn nicht, wird eine „erfundene“ Scheibe hinzugefügt. Diese erfundenen Scheiben werden z.B. in einer Menge gespeichert. Wenn die Anzahl der erfundenen Scheiben der Anzahl der entfernten Scheiben entspricht, kommen keine neuen mehr hinzu.

Ergänzung zur Zeitkomplexität:

Die Anzahl der gegessenen Scheiben hat grundsätzlich keine Auswirkungen auf die Zeitkomplexität, eher auf die durchschnittliche Laufzeit. Dadurch, dass es nun Scheiben mit variabler Größe gibt, können mehr verschiedene Quader konstruiert werden, die Laufzeit nähert sich also O(3^n) an.

Umsetzung

Ich habe den Hauptteil der Aufgabe in Rust und einzelne Teile, wie das Erstellen eines Tests für den Aufgabenteil B, in Python implementiert. Die Einzelteile werden in einem Bash script zusammengeführt (das Programm kann nur ausgeführt werden, wenn run.sh, solve, und solve2 executable permissions bekommen, ich habe das Ganze auf 2 verschiedenen Geräten mit Linux ausprobiert)

Die read_input und die solve function im Quellcode Abschnitt stammen aus Programm B.

In read_input wird der erste Teil des Problems gelöst. Als erstes wird das Input File eingelesen und die Koordinaten zu f64 geparst. Die Variable lowest hält den bisher geringsten Wert für z (aus Lösungsidee 1.), das HashSet lowest_second hält alle möglichen Werte für x und y. In all_slices sind alle Scheiben gespeichert.

solve startet damit, alle möglichen Anfangs-Scheiben herauszusuchen, in jedem Durchgang der äußeren Schleife ist eine dieser Scheiben der Ausgangspunkt für den backtracking Algorithmus. Die zwischendurch gegessenen Scheiben werden in made_up_slices gespeichert. Wie und wo eine Käsescheibe an den Quader angelegt wurde, wird gespeichert als ([bool; 3], usize, bool), der Array an Index 0 enthält, welche Möglichkeiten, an einen bestimmten Quader Scheiben anzulegen, schon ausprobiert wurden, also ob die Scheibe die Längen [x, y], [x, z] oder [y, z] hatte. An index 1 wird gespeichert, welche dieser Varianten ausprobiert wird oder wurde und in Index 2, ob die Scheibe aus all_slices war oder ob sie zu made_up_slices gehört. Diese Struktur wird nun verwendet, um herauszufinden, welche der drei Möglichkeiten, eine Scheibe auszuwählen und an den Quader zu legen in der Reihenfolge als nächstes kommt, nachdem back_step true war. Grundsätzlich wird, wenn eine Scheibe auf den Quader gelegt wird, diese Struktur auf den stack gelegt und bei einem backstep abgenommen. Die Funktion order sortiert x, y und z nach Größe.

Bei Aufgabe B werden die Beispiele aus der Aufgabe modifiziert verwendet. Das Python Script, das die Scheiben entfernt, nimmt dazu die ungeordnete Liste als Basis und entfernt Scheiben, die nicht zum Kern des Käsequaders gehören. Dafür wird auf den Output von Aufgabe A zugegriffen.

Wenn eine Käsescheibe entfernt wird, die sonst als letztes an den Quader angelegt werden würde, ist der Quader komplett, auch ohne diese Scheibe.

Für Aufgabenteil A und B gibt es je einen testcases und einen output folder mit files.

Beispiele

Aufgabe A

```
$ ./run.sh
Program B oder A: A
"all" fuer alle Beispiele oder name eines files im testcase folder: all
bsp1.txt
status: success
Time per test: 183.956µs
```

```
number of slices: 12
Time per slice in nanosecs: 15329
bsp2.txt
status: success
Time per test: 64.173µs
number of slices: 5
Time per slice in nanosecs: 12834
bsp3.txt
status: success
Time per test: 72.626µs
number of slices: 23
Time per slice in nanosecs: 3157
bsp4.txt
status: success
Time per test: 520.418µs
number of slices: 550
Time per slice in nanosecs: 946
bsp5.txt
status: success
Time per test: 4.62536ms
number of slices: 6848
Time per slice in nanosecs: 675
bsp6.txt
status: success
Time per test: 25.376516ms
number of slices: 90300
Time per slice in nanosecs: 281
bsp7.txt
status: success
Time per test: 523.75602ms
number of slices: 1529903
Time per slice in nanosecs: 342
Total time for all tests: 554.667571ms

für kaese 1 (output/bsp1.txt, die restlichen Output files sind auch im output
folder)
2, 4
2, 4
2, 4
3, 4
3, 3
3, 3
3, 6
4, 6
4, 6
4, 6
6, 6
6, 6
```

Aufgabe B

```
$ ./run.sh
Program B oder A: B
"all" fuer alle Beispiele oder name eines files im testcase folder: all
Anzahl der gegessenen Kaesescheiben: 3
removed:
```

```
 ['3 3\n', '3 3\n', '6 6\n']
removed:
 ['2 1000\n', '2 1000\n', '2 998\n']
removed:
 ['4 998\n', '2 997\n', '999 1000\n']
removed:
 ['171 174\n', '12 38\n', '195 196\n']
removed:
 ['2193 3053\n', '2190 3050\n', '335 1645\n']
removed:
 ['39110 510356\n', '7379 487498\n', '15989 25239\n']
removed:
 ['493499 493880\n', '191234 191630\n', '223958 224705\n']

Test bsp1.txt
path: testcasesb/bsp1.txt
removed:
{[3, 3]: 2}
success
time for one testcase: 126.428µs

Test bsp2.txt
path: testcasesb/bsp2.txt
time for one testcase: 5.363µs

Test bsp3.txt
path: testcasesb/bsp3.txt
removed:
{[2, 997]: 1, [999, 1000]: 1, [4, 998]: 1}
success
time for one testcase: 69.456µs

Test bsp4.txt
path: testcasesb/bsp4.txt
removed:
{[171, 174]: 1, [12, 38]: 1, [195, 196]: 1}
success
time for one testcase: 194.382µs

Test bsp5.txt
path: testcasesb/bsp5.txt
removed:
{[2193, 3053]: 1, [2190, 3050]: 1, [335, 1645]: 1}
success
time for one testcase: 5.064129ms

Test bsp6.txt
path: testcasesb/bsp6.txt
removed:
{[7379, 487498]: 1, [39110, 510356]: 1, [15989, 25239]: 1}
success
time for one testcase: 53.931509ms

Test bsp7.txt
path: testcasesb/bsp7.txt
removed:
{[223958, 224705]: 1, [191234, 191630]: 1, [493499, 493880]: 1}
```

```
success
time for one testcase: 586.27119ms
Total time for all Testcases: 645.688883ms
```

Quellcode

```
fn read_input(path: &str) -> (i32, HashSet<usize>, HashMap<[usize; 2], usize>, i64) {
    println!("path: {}", path);
    // read input file
    let file: File = File::open(path).unwrap();
    let mut all_slices: HashMap<[usize; 2], usize> = HashMap::new();
    let reader = BufReader::new(file);
    let mut lines = reader.lines();
    let n: i32 = lines.next().unwrap().unwrap().parse::<i32>().unwrap();
    let mut lowest: usize = n as usize;
    let mut lowest_second: HashSet<usize> = HashSet::new();
    for line in lines { // iterate over all slices
        let line: String = line.unwrap();
        let mut line2 = line.trim().split_whitespace();
        let n1: usize = usize::from_str(line2.next().unwrap()).unwrap();
        let n2: usize = usize::from_str(line2.next().unwrap()).unwrap();
        // new lowest
        if n1 < lowest {
            lowest = n1;
            lowest_second.clear();
        }
        // new lowest_second
        if n1 == lowest {
            lowest_second.insert(n2);
        }
        let nums: [usize; 2] = [n1, n2];
        *all_slices.entry(nums).or_insert(0) += 1;
    }
    (lowest as i32, lowest_second, all_slices, n as i64)
}

fn solve(
    lowest: usize,
    lowest_second: HashSet<usize>,
    mut all_slices: HashMap<[usize; 2], usize>,
    file_name: &str,
    number_eaten: usize,
) {
    // indentify possible start slices
    let number_eaten: usize = number_eaten;
    let mut start_slices: Vec<[usize; 2]> = Vec::new(); // change all_slices to mut
    for (key, value) in all_slices.iter() {
        if *value >= lowest && (lowest_second.contains(&key[0]) ||
lowest_second.contains(&key[1]))
        {
            start_slices.push(key.clone());
        }
    }
    while !start_slices.is_empty() { // iterate over all possible start slices
        let start: [usize; 2] = start_slices.pop().unwrap();
        let mut x: usize = start[0];
        let mut y: usize = start[1];
        let mut z: usize = lowest;
        modify_hash_map(&mut all_slices, start, -(lowest as i32));
        let mut stack: Vec<([bool; 3], usize, bool)> = Vec::new();
        let mut step_backward: bool = false;
        order(&mut x, &mut y, &mut z);
        let mut state: ([bool; 3], usize, bool);
        let mut success: bool = true;
        let mut made_up_slices: HashMap<[usize; 2], usize> = HashMap::new();
```

```
        let mut made_up_slices_count: usize = 0;
        // Backtracking Algorithm
        while !all_slices.is_empty() || step_backward {
            if step_backward { // pop last slice
                order(&mut x, &mut y, &mut z);
                if stack.is_empty() { // no solution
                    success = false;
                    break;
                }
                state = stack.pop().unwrap();
                // modify x, y or z
                let slice = back_backtrack(state.1, &mut x, &mut y, &mut z);
                // reconstruct slice
                if !state.2 {
                    modify_hash_map(&mut all_slices, slice, 1); //t
                } else {
                    modify_hash_map(&mut made_up_slices, slice, -1);
                    made_up_slices_count -= 1;
                }
                if state.0 == [true, true, true] { // all slices tried for this cube ->
continue backstep
                    continue;
                } else { // backward to false
                    step_backward = false;
                }
            } else {
                let empty: [bool; 3] = [false, false, false];
                state = (empty, 0, false);
            }
            // push new slice
            order(&mut x, &mut y, &mut z);
            let arr: [bool; 3] = [
                all_slices.contains_key(&[x, y]),
                all_slices.contains_key(&[x, z]),
                all_slices.contains_key(&[y, z]),
            ];
            let mut found_slice: bool = false;
            // look for fitting slice in all_slices
            if !state.2 {
                if !state.0[0] && arr[0] {
                    state.0[0] = true;
                    state.1 = 1;
                    stack.push(state);
                    z += 1;
                    found_slice = true;
                    modify_hash_map(&mut all_slices, [x, y], -1)
                } else if !state.0[1] && arr[1] {
                    state.0[1] = true;
                    state.1 = 2;
                    y += 1;
                    if y > z {
                        state.1 = 1;
                    }
                    stack.push(state);
                    found_slice = true;
                    modify_hash_map(&mut all_slices, [x, z], -1);
                } else if !state.0[2] && arr[2] {
                    state.0[2] = true;
                    state.1 = 3;
                    x += 1;
                    if x > y {
                        state.1 = 2;
                        if x > z {
                            state.1 = 1;
                        }
                    }
```

```
                    stack.push(state);
                    found_slice = true;
                    modify_hash_map(&mut all_slices, [y, z], -1);
                } else {
                    state.2 = true;
                }
            } // if no fitting slice found try make one up
            if state.2 && made_up_slices_count < number_eaten {
                made_up_slices_count += 1;
                if !state.0[0] {
                    state.0[0] = true;
                    state.1 = 1;
                    stack.push(state);
                    z += 1;
                    found_slice = true;
                    modify_hash_map(&mut made_up_slices, [x, y], 1)
                } else if !state.0[1] {
                    state.0[1] = true;
                    state.1 = 2;
                    y += 1;
                    if y > z {
                        state.1 = 1;
                    }
                    stack.push(state);
                    found_slice = true;
                    modify_hash_map(&mut made_up_slices, [x, z], 1);
                } else if !state.0[2] {
                    state.0[2] = true;
                    state.1 = 3;
                    x += 1;
                    if x > y {
                        state.1 = 2;
                        if x > z {
                            state.1 = 1;
                        }
                    }
                    stack.push(state);
                    found_slice = true;
                    modify_hash_map(&mut made_up_slices, [y, z], 1);
                } else {
                    made_up_slices_count -= 1;
                }
            // modify step_backwards
            } else {
                step_backward = true;
            }
            if found_slice {
                step_backward = false;
            }
        }
        if success {
            // output found solution
            output_rev(stack, start, lowest, file_name);
            if PRINT_REMOVED {
                print_removed(made_up_slices);
            }
            return;
        } else {
            modify_hash_map(&mut all_slices, start, lowest as i32);
        }
    }
}
```