

# Datastructures: opdracht 2

---

Docent: Gerke de Boer

---

Duy Tran 500668322

---

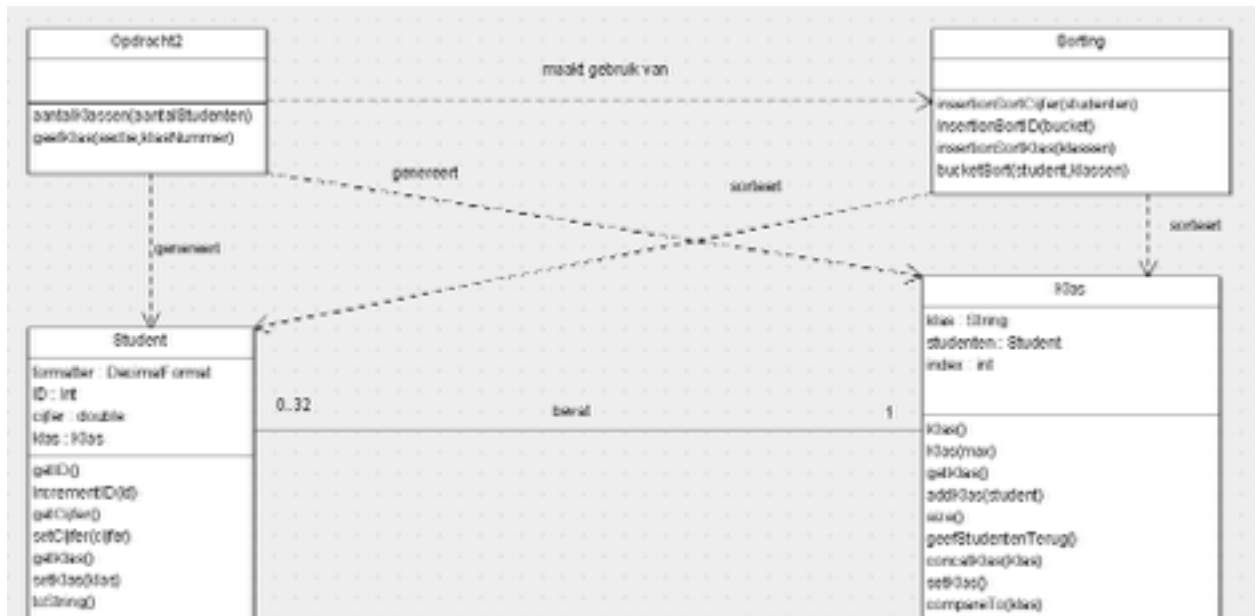
Tzemo Chan 500637662

Klassediagram	3
Uitleg programma + code snippets	3
Grafieken	12
Tabellen:	12
Insertion Sort op cijfers.	13
Bucket Sort.	13

## Klassediagram

In het verslag dat wij moesten opleveren dient een klassediagram bij te zitten en de uitleg van onze programma:

Hier de bijbehorende klassediagram voor onze programma.



## Uitleg programma + code snippets

Opdracht2 is onze main klasse die operaties uitvoert zoals het aantal klassen genereren en ze aanmaken. In de main methode worden ook de studenten bijvoorbeeld aangemaakt.

Hier een voorbeeld code hoe de studenten worden aangemaakt:

```
//Maak studenten aan
for (int i = 0; i < studenten.length; i++) {
    Student s = new Student();
    s.incrementId(i);
    double cijfer = generator.nextDouble() * 9 + 1;
    s.setCijfer(cijfer);
    studenten[i] = s;
}
```

We maken gebruik van een for-loop om alle studenten aan te maken en ze vervolgens een ID te geven en een random cijfer van 1.0 of 10.0, maar er mist nog iets. Ze hebben nog geen klas! In de onderstaande code worden de studenten een klasse aangewezen.

```
counter = 0;

//Voeg student toe aan de klassen.
for (int i = 0; i < studenten.length; i++) {
    if (counter < klassen.size()) {
        studenten[i].setKlas(klassen.get(counter));
        klassen.get(counter).addStudent(studenten[i]);
        counter++;
        if (counter == klassen.size()) {
            counter = 0;
        }
    }
}
```

Voordat deze code wordt uitgevoerd hebben we eerst de klassen aangemaakt op basis van het aantal studenten. De klassen bevinden zich in een ArrayList. De counter variabele is ervoor om ervoor te zorgen dat er telkens door de klassen ArrayList heen geloopt wordt zodat de studenten erin worden geplaatst. Als de counter gelijk staat aan de size van de ArrayList, dan wordt ie weer naar 0 gezet zodat je er weer door heen kan gaan.

Er is nu een klein probleem. We hebben nu een ArrayList van klassen die niet compleet vol zitten met studenten. In de klasse Klas zit een attribuut array genaamd studenten die een vaste waarde heeft van 32. Als ik 200 studenten heb zal er maar 25 in een klas zitten, dus ik zit dan vast met een array die Null waardes bevat.

Hieronder bevindt onze oplossing daarvoor.

We hebben de constructor van Klas geoverload:

```
public Klas(int max) {

    if (max <= 32) {
        studenten = new Student[max];
    } else {
        System.out.println("Error");
    }
    index = 0;
}
```

Als je een Klas aanmaakt heb je alvast de mogelijkheid om aan te geven hoeveel studenten erin gaan komen.

Hieronder de uiteindelijke oplossing.

```

ArrayList<Klas> legeKlassen = new ArrayList();

//Lijst zonder null waarden.
for (int i = 0; i < klassen.size(); i++) {

    Klas current = klassen.get(i);
    Student[] s = current.geefStudentenTerug();

    Klas leeg = new Klas(current.size());
    leeg.setKlas(current.getKlas());

    legeKlassen.add(leeg);

}

```

We maken een nieuwe ArrayList aan genaamd legeKlassen. Vervolgens lopen we door de klassen heen en maken we een object van Klas genaamd current. Vanuit current halen we de bijbehorende array op. Daarna maken we een nieuwe object van Klas genaamd leeg en in de constructor geven we current.size() mee. current.size() is NIET de lengte van de array, maar geeft het aantal elementen terug van de array die we bijhouden als we telkens een studente toevoegen. Als laatst zetten we de lege klas in de ArrayList.

Nu gaan de output tonen van de studenten voor en nadat ze geshuffled zijn. Links bevindt de lijst van de studenten voor de shuffle en bij rechts staat de geshuffelde lijst.

Student{id=50060001, cijfer=8.6, klas=IG201}	SHUFFLE
Student{id=50060002, cijfer=9.1, klas=IN201}	Student{id=50060380, cijfer=5.1, klas=IT220}
Student{id=50060003, cijfer=9.2, klas=IS201}	Student{id=50062845, cijfer=2.9, klas=IG212}
Student{id=50060004, cijfer=1.9, klas=IT201}	Student{id=50062446, cijfer=7.7, klas=IN212}
Student{id=50060005, cijfer=7.4, klas=IG202}	Student{id=50062930, cijfer=4.7, klas=IN208}
Student{id=50060006, cijfer=1.8, klas=IN202}	Student{id=50060725, cijfer=9.8, klas=IG207}
Student{id=50060007, cijfer=9.8, klas=IS202}	Student{id=50062844, cijfer=1.3, klas=IT211}
Student{id=50060008, cijfer=9.8, klas=IT202}	Student{id=50063001, cijfer=6.9, klas=IG201}
Student{id=50060009, cijfer=1.2, klas=IG203}	Student{id=50060558, cijfer=4.2, klas=IN215}
Student{id=50060010, cijfer=2.1, klas=IN203}	Student{id=50063131, cijfer=4.6, klas=IS208}
Student{id=50060011, cijfer=5.7, klas=IS203}	Student{id=50062210, cijfer=3.4, klas=IN203}
Student{id=50060012, cijfer=2.0, klas=IT203}	Student{id=50061228, cijfer=3.5, klas=IT207}
Student{id=50060013, cijfer=6.3, klas=IG204}	Student{id=50060649, cijfer=6.7, klas=IG213}
Student{id=50060014, cijfer=2.6, klas=IN204}	Student{id=50061516, cijfer=3.9, klas=IT204}
Student{id=50060015, cijfer=5.6, klas=IS204}	Student{id=50060228, cijfer=5.9, klas=IT207}
Student{id=50060016, cijfer=7.2, klas=IT204}	Student{id=50062310, cijfer=2.5, klas=IN203}
Student{id=50060017, cijfer=5.9, klas=IG205}	Student{id=50061397, cijfer=2.9, klas=IG225}
Student{id=50060018, cijfer=5.0, klas=IN205}	Student{id=50062936, cijfer=6.0, klas=IT209}
Student{id=50060019, cijfer=7.9, klas=IS205}	Student{id=50060898, cijfer=4.5, klas=IN225}
Student{id=50060020, cijfer=5.5, klas=IT205}	Student{id=50062397, cijfer=7.1, klas=IG225}
Student{id=50060021, cijfer=7.5, klas=IG206}	Student{id=50062894, cijfer=1.9, klas=IN224}
Student{id=50060022, cijfer=6.6, klas=IN206}	Student{id=50062715, cijfer=4.4, klas=IS204}
Student{id=50060023, cijfer=6.3, klas=IS206}	Student{id=50060505, cijfer=7.7, klas=IG202}
Student{id=50060024, cijfer=3.1, klas=IT206}	Student{id=50061962, cijfer=2.3, klas=IN216}
Student{id=50060025, cijfer=9.4, klas=IG207}	Student{id=50061069, cijfer=5.7, klas=IG218}
Student{id=50060026, cijfer=4.6, klas=IN207}	Student{id=50061232, cijfer=9.1, klas=IN208}
Student{id=50060027, cijfer=7.5, klas=IS207}	Student{id=50060212, cijfer=1.4, klas=IT203}
Student{id=50060028, cijfer=1.8, klas=IT207}	Student{id=50060820, cijfer=5.7, klas=IT205}
	Student{id=50060662, cijfer=1.3, klas=IN216}
	Student{id=50060565, cijfer=9.4, klas=IG217}
	Student{id=50060801, cijfer=3.0, klas=IG201}

Wat we nu allereerst gaan doen is een insertion sort uitvoeren om de cijfers van hoog naar laag te sorteren.

Dit is de code die we gebruikt hadden om dat te realiseren:

```
public static Student[] insertionSortCijfer(Student[] studenten) {

    double t1 = System.nanoTime();
    System.out.println("Aantal studenten: " + studenten.length);

    for (int i = 1; i < studenten.length; i++) {

        Student temp = studenten[i];
        int j = i;

        while (j > 0 && studenten[j - 1].getCijfer() < temp.getCijfer()) {

            studenten[j] = studenten[j - 1];

            j--;
        }

        studenten[j] = temp;
    }

    double t2 = System.nanoTime();

    System.out.println("Tijd benodigd = " + (t2-t1) / 1000000000);

    return studenten;
}
```

Allereerst wat we hier doen is de tijd opmeten.

Vervolgens printen we uit met hoeveel studenten we te maken hebben.

Na het uitprinten daarvan beginnen we met de loop. Onze loop begint bij 1 omdat tijdens de while loop terug gaan kijken in de array, dus als we bij index 0 beginnen en we kijken gelijk terug, dan krijgen we een `ArrayOutOfBoundsException`.

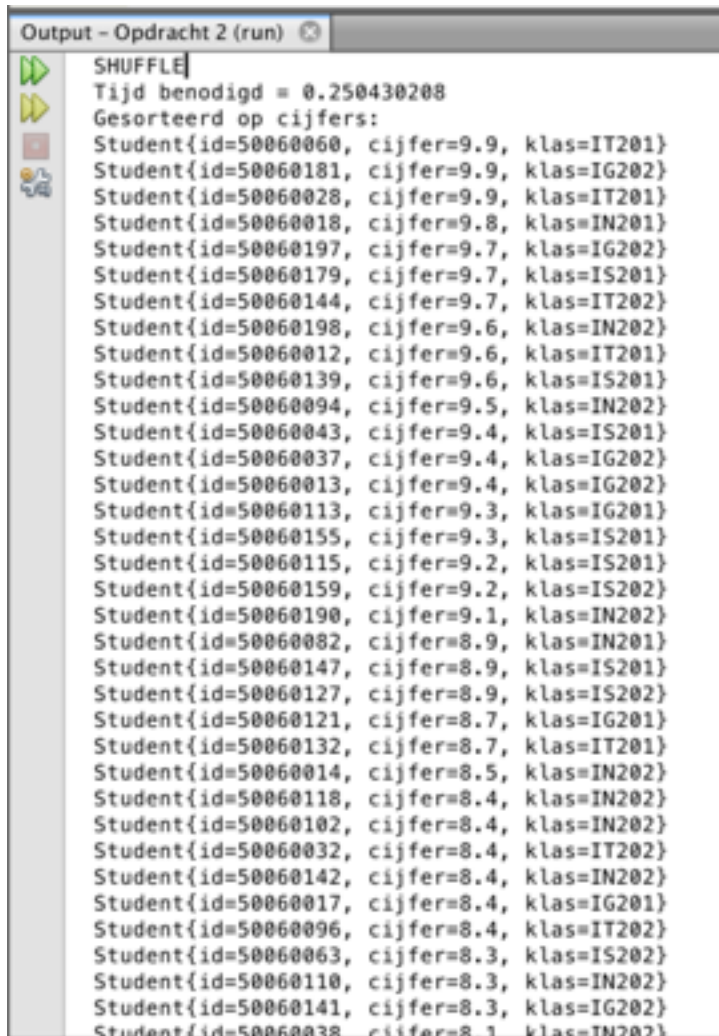
In de loop slaan we de huidige Student op in het object 'temp'. Vervolgens slaan we i op in int j, want j gaan we gebruiken als index om telkens terug te kijken.

We hebben een while loop ingezet dat checkt of j groter is dan 0 en als het cijfer van de vorige student kleiner is dan die van temp dan swappen we ze en zetten we de grotere bovenaan.

Als we uit de for-loop zijn, nemen we de tijd op dat moment op en printen we het verschil uit tussen t-1 en t-2.

Uiteindelijk geven de studenten array weer terug.

Hier de output voor de insertion sort op cijfers:



```
SHUFFLE
Tijd benodigd = 0.250430208
Gesorteerd op cijfers:
Student{id=50060060, cijfer=9.9, klas=IT201}
Student{id=50060181, cijfer=9.9, klas=IG202}
Student{id=50060028, cijfer=9.9, klas=IT201}
Student{id=50060018, cijfer=9.8, klas=IN201}
Student{id=50060197, cijfer=9.7, klas=IG202}
Student{id=50060179, cijfer=9.7, klas=IS201}
Student{id=50060144, cijfer=9.7, klas=IT202}
Student{id=50060198, cijfer=9.6, klas=IN202}
Student{id=50060012, cijfer=9.6, klas=IT201}
Student{id=50060139, cijfer=9.6, klas=IS201}
Student{id=50060094, cijfer=9.5, klas=IN202}
Student{id=50060043, cijfer=9.4, klas=IS201}
Student{id=50060037, cijfer=9.4, klas=IG202}
Student{id=50060013, cijfer=9.4, klas=IG202}
Student{id=50060113, cijfer=9.3, klas=IG201}
Student{id=50060155, cijfer=9.3, klas=IS201}
Student{id=50060115, cijfer=9.2, klas=IS201}
Student{id=50060159, cijfer=9.2, klas=IS202}
Student{id=50060190, cijfer=9.1, klas=IN202}
Student{id=50060082, cijfer=8.9, klas=IN201}
Student{id=50060147, cijfer=8.9, klas=IS201}
Student{id=50060127, cijfer=8.9, klas=IS202}
Student{id=50060121, cijfer=8.7, klas=IG201}
Student{id=50060132, cijfer=8.7, klas=IT201}
Student{id=50060014, cijfer=8.5, klas=IN202}
Student{id=50060118, cijfer=8.4, klas=IN202}
Student{id=50060102, cijfer=8.4, klas=IN202}
Student{id=50060032, cijfer=8.4, klas=IT202}
Student{id=50060142, cijfer=8.4, klas=IN202}
Student{id=50060017, cijfer=8.4, klas=IG201}
Student{id=50060096, cijfer=8.4, klas=IT202}
Student{id=50060063, cijfer=8.3, klas=IS202}
Student{id=50060110, cijfer=8.3, klas=IN202}
Student{id=50060141, cijfer=8.3, klas=IG202}
Student{id=50060038, cijfer=8.1, klas=IN202}
```

En dat werkt ook.

Nu moeten we de studenten sorteren op klas en in de klas moeten we ze ook sorteren op studentnummer.

Daarvoor maken we gebruik van bucket sort.

```
public static Student[] bucketSort(Student[] studenten, ArrayList klassen) {

    double t1 = System.nanoTime();

    Klas[] buckets = new Klas[klassen.size()];

    for (int i = 0; i < klassen.size(); i++) {
        buckets[i] = (Klas) klassen.get(i);
    }
}
```

```

buckets = insertionSortKlas(buckets);

for (int i = 0; i < buckets.length; i++) {

    for (int j = 0; j < studenten.length; j++) {

        if (studenten[j].getKlas().compareTo(buckets[i]) == 0) {
            buckets[i].addStudent(studenten[j]);
        }

    }

}

for (int i = 0; i < buckets.length; i++) {
    buckets[i] = insertionSortID(buckets[i]);
}

int index = 0;

for (int i = 0; i < buckets.length; i++) {

    Student[] s = buckets[i].geefStudentenTerug();

    for (int j = 0; j < s.length; j++) {
        studenten[index] = s[j];
        index++;
    }

}

double t2 = System.nanoTime();

System.out.println("Tijd benodigd: " + (t2-t1) / 1000000000);

return studenten;

}

```

Onze bucket sort methode heeft 2 argumenten nodig.

De eerste argument is de array van studenten en de 2e argument is een ArrayList van klassen.

Allereerst nemen we de tijd op en daarna maken we een array aan van buckets waarvan de grootte bepaald wordt door de size van de ArrayList klassen.

Daarna lopen we door de ArrayList heen en zetten we elke index van de ArrayList in de bucket waardoor de bucket nu de klassen bevat.



Daarna sorteren we de buckets zodat de klassen op volgorde staan.

Als dat gedaan is lopen we door de buckets heen en tijdens elke bucket lopen we door alle studenten heen en maken gebruik van de compareTo methode om te checken of de klas van de Student gelijk staat aan die van de huidige bucket. Als dat gelijk staat aan 0, dan stoppen we de student in de bucket.

Als dat gedaan is, dan lopen we weer door de buckets heen, maar dit keer gaan per bucket alle studenten op studentnummer sorteren via de insertion sort method die we hebben aangemaakt.

```
public static Klas insertionSortID(Klas bucket) {
    Student[] s = bucket.geefStudentenTerug();

    Klas k = new Klas(bucket.geefStudentenTerug().length);

    for (int i = 0; i < s.length ; i++) {
        Student temp = s[i];
        int j = i;

        while (j > 0 && s[j - 1].getId() > temp.getId()) {

            s[j] = s[j - 1];

            j--;
        }

        s[j] = temp;
    }

    for (int i = 0; i < s.length; i++) {
        k.addStudent(s[i]);
    }

    return k;
}
```

Deze methode is bijna hetzelfde als die van de insertion sort voor de cijfers alleen hier geven we een Klas object terug met daarbij de gesorteerde leerlingen erin. Als dat afgemaakt is komen we bij de laatste for-loop van onze code:

```
int index = 0;

for (int i = 0; i < buckets.length; i++) {

    Student[] s = buckets[i].geefStudentenTerug();

    for (int j = 0; j < s.length; j++) {
        studenten[index] = s[j];
        index++;
    }

}
```

We lopen wederom weer door de buckets heen en we halen de studenten terug uit elke bucket. In de for-loop zit nog een for-loop die alle studenten toevoegt van de bucket. De index is om ervoor te zorgen dat we telkens naar de volgend element gaan. Na dat dit is gedaan stoppen we de tijd en berekenen we dat en daarna geven we de studenten array terug gesorteerd op klas en studentnummer.

Nu gaan de output laten zien voor de bucket sort:

```
Gesorteerd op ID en klas
Aantal studenten:200
Tijd benodigd: 0.00375808
Student{id=50060001, cijfer=3.0, klas=IG201}
Student{id=50060009, cijfer=6.4, klas=IG201}
Student{id=50060017, cijfer=9.6, klas=IG201}
Student{id=50060025, cijfer=6.7, klas=IG201}
Student{id=50060033, cijfer=3.3, klas=IG201}
Student{id=50060041, cijfer=9.8, klas=IG201}
Student{id=50060049, cijfer=8.6, klas=IG201}
Student{id=50060057, cijfer=7.0, klas=IG201}
Student{id=50060065, cijfer=3.2, klas=IG201}
Student{id=50060073, cijfer=6.5, klas=IG201}
Student{id=50060081, cijfer=6.0, klas=IG201}
Student{id=50060089, cijfer=9.2, klas=IG201}
Student{id=50060097, cijfer=2.5, klas=IG201}
Student{id=50060105, cijfer=4.2, klas=IG201}
Student{id=50060113, cijfer=4.0, klas=IG201}
Student{id=50060121, cijfer=9.5, klas=IG201}
Student{id=50060129, cijfer=6.9, klas=IG201}
Student{id=50060137, cijfer=6.8, klas=IG201}
Student{id=50060145, cijfer=4.3, klas=IG201}
Student{id=50060153, cijfer=2.7, klas=IG201}
Student{id=50060161, cijfer=6.5, klas=IG201}
Student{id=50060169, cijfer=1.9, klas=IG201}
Student{id=50060177, cijfer=5.5, klas=IG201}
Student{id=50060185, cijfer=7.4, klas=IG201}
Student{id=50060193, cijfer=1.9, klas=IG201}
Student{id=50060005, cijfer=3.4, klas=IG202}
Student{id=50060013, cijfer=4.7, klas=IG202}
Student{id=50060021, cijfer=6.6, klas=IG202}
Student{id=50060029, cijfer=6.2, klas=IG202}
Student{id=50060037, cijfer=6.4, klas=IG202}
Student{id=50060045, cijfer=5.2, klas=IG202}
Student{id=50060053, cijfer=2.8, klas=IG202}
Student{id=50060061, cijfer=1.8, klas=IG202}
Student{id=50060069, cijfer=8.8, klas=IG202}
Student{id=50060077, cijfer=2.5, klas=IG202}
Student{id=50060085, cijfer=3.6, klas=IG202}
Student{id=50060093, cijfer=1.7, klas=IG202}
Student{id=50060101, cijfer=5.5, klas=IG202}
Student{id=50060109, cijfer=2.3, klas=IG202}
Student{id=50060117, cijfer=7.0, klas=IG202}
```

---

Zoals je kan zien zijn de studenten gesorteerd op klas en vervolgens op studentnummer.

## Grafieken

## Tabellen:

Aantal Studenten	Sorteer op cijfer (Tijd in seconde)	Sorteer op ID en Klas (Tijd in seconde)
12800	48,205034453	0,882265856
6400	12,86626767	0,458350848
3200	3,718410067	0,118191104
1600	0,998164034	0,037159936
800	0.320607068	0,1252992
400	0.150897702	0,01421312
200	0.096911621	0,00375808

Uit de gegevens van de tabellen komen wij tot de conclusie dat de bunkert sort veel efficiënter is dan de insertion sort.

## **Insertion Sort op cijfers.**

Wat ons is opgevallen bij de tijden van het sorteren op cijfers is dat wanneer het aantal studenten verdubbeld dat de tijd dan verviervoudigd.

De factor die we daaruit halen is:  $\sim 4$

De berekening die we hiervoor hebben gebruikt is de tijd van het aantal studenten delen door de vorige tijd van het vorige aantal studenten. (Bijv: de tijd van 12800 / tijd van 6400 = 3,746621451487337 en daarna tijd van 6400 / tijd van 3200 enzovoort.).

## **Bucket Sort.**

Bij de kleinere getallen is het zo snel dat het bijna niet te meten valt. De waardes verschillen heel erg.