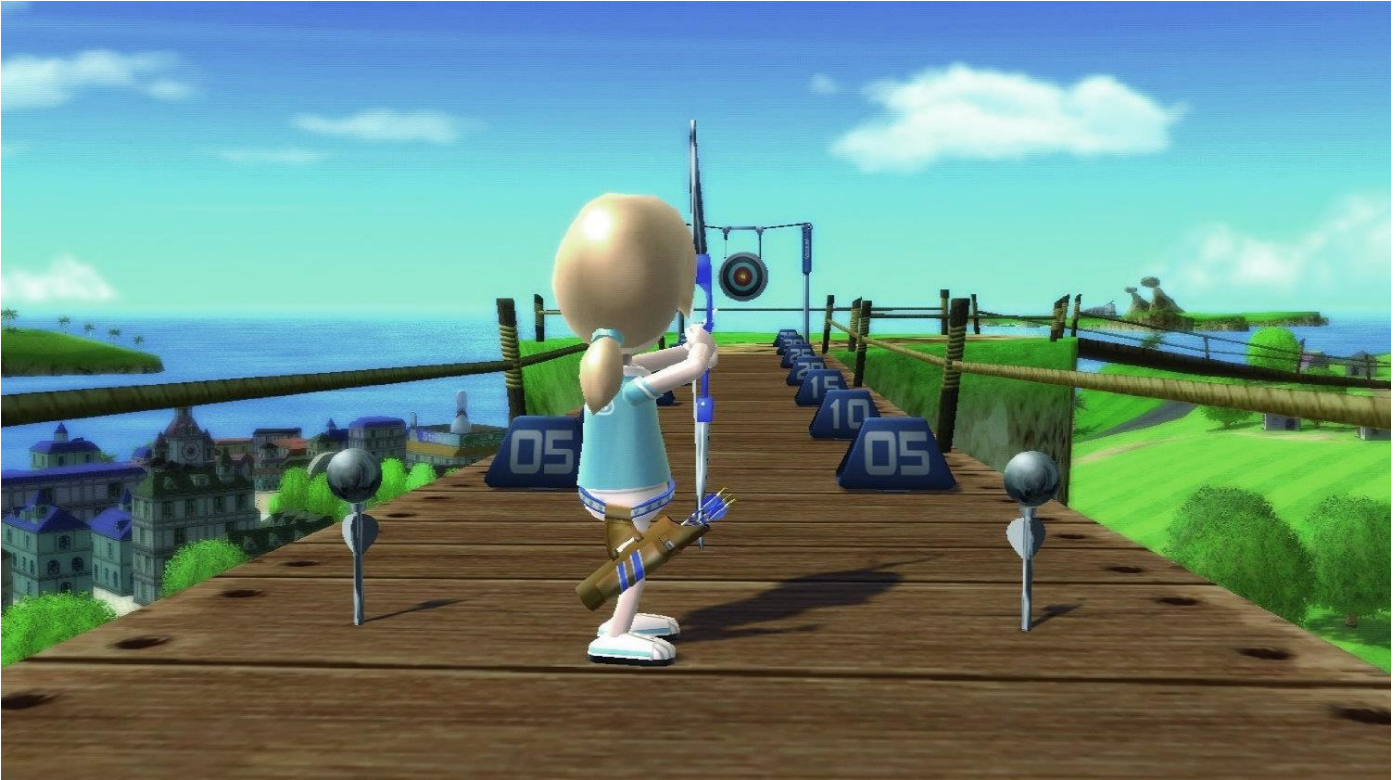


Archery

Verslag over programmeeropdracht archery



Tijs de Graaff (500802245)

Auke Steenman (500800707)

Is206 | versie 1 (6 oktober, 2019)

Inhoudsopgave

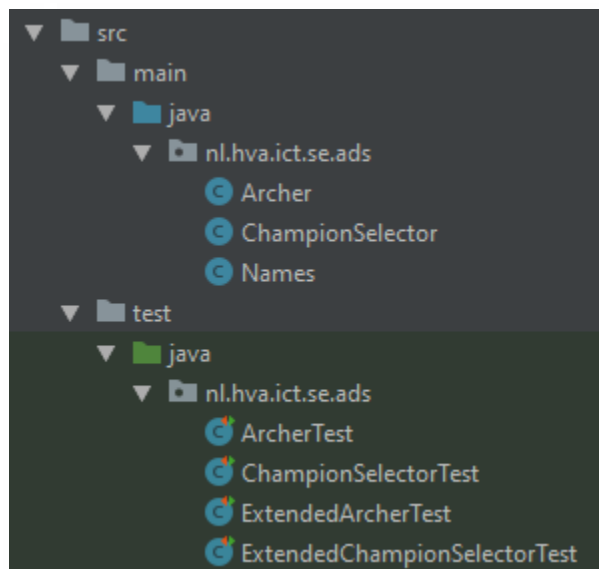
| | |
|---------------------------------------|----|
| Introductie | 3 |
| Classes..... | 3 |
| Archer..... | 3 |
| ChampionSelector..... | 4 |
| Names | 4 |
| Opdrachten..... | 5 |
| Genereer boogschutters en scores..... | 5 |
| Onveranderlijk boogschutter id | 5 |
| Uniek boogschutter id..... | 5 |
| Boogschutter toString methode | 6 |
| Voeg het score systeem toe..... | 7 |
| Vind de kampioen | 9 |
| Efficiënt sorteren | 11 |
| Insertion sort..... | 12 |
| Quick sort | 12 |
| Java collection sort..... | 13 |

Introductie

Dit document is opgesteld om meer informatie te geven hoe wij, Tijs en Auke de opdracht *Archery* hebben gemaakt. In dit document beschrijven we de denkwijze en tegen welke problemen wij zijn aangelopen. We zullen beginnen met het beschrijven van classes en functionaliteit. Hierna zullen we de sub-opdrachten bespreken van de assignment. Hier gaan wij onze belangrijkste code laten zien met toelichting. In de conclusie staat wat wij vonden van deze opdracht en feedback op de opdracht. De opdracht gaat over het bijhouden van een boogschutter competitie.

Classes

Het project bestaat uit 3 classes: **Archer**, **ChampionSelector** en **Names**. De belangrijkste classes van dit project zijn **Archer** en **ChampionSelector**, naar deze classes zullen we vaker refereren door dit verslag heen. Voor een snelle overview van de structuur van het project (zie Figuur 1).



Figuur 1: Project structuur

Archer

De archer refereert naar een boogschutter in het toernooi. Hier worden de scores van de boogschutter bijgehouden in vorm van matrix array (`int[][]`), maar ook de naam van de boogschutter en nog een aantal andere utility features zoals randomizer voor scores en het maximaal aantal pijlen en rondes relevant voor de boogschutter.

ChampionSelector

Deze utility class bevat allerlei manieren om een lijst van boogschutters te sorteren. Elke functie staat gelijk aan een sorteer algoritme.

Names

Deze utility class bevat een gigantische lijst van voornamen en achternamen en een randomizer om hier willekeurige waardes uit op te vragen.

Opdrachten

De opdrachten zijn ingedeeld in een aantal fases:

- Genereer boogschutters en scores
- Vind de kampioen
- Efficiënt sorteren

Genereer boogschutters en scores

In deze opdracht vullen we de missende code snippets in en zorgen we dat het genereren van boogschutters en bijbehorende scores naar genoeg werkt.

Onveranderlijk boogschutter id

Het eerste wat gevraagd word is om na het toewijzen van een id van een boogschutter, deze onveranderbaar te maken. Dit zorgt ervoor dat een boogschutter geen ander id toegewezen kan krijgen nadat deze is aangemaakt. Dit kleine probleem heb ik simpelweg opgelost door het **final** toe te passen op het id.

```
1. private final int id; // Once assigned a value is not allowed to change.
```

“The **final** keyword is used in several contexts to define an entity that can only be assigned once.” – Wikipedia artikel final (Java)

Uniek boogschutter id

Daarna werd er gevraagd om elk uniek id van een boogschutter oplopend toe te wijzen voor elke nieuwe boogschutter (de eerste boogschutter begint met id 135788). Dit heb ik opgelost door een property toe te voegen aan de constructor van een boogschutter. Door een **int id** mee te geven aan een boogschutter, kan ik het id manipuleren vanuit de loop waarin boogschutter worden aangemaakt. Ook houd ik een class constante bij met de waarde van het begin id in dezelfde stijl van MAX_ARROWS en MAX_ROUNDS.

```
1. for (int i = 0; i < nrOfArchers; i++) {  
2.     Archer archer = new Archer(  
3.         Names.nextFirstName(),  
4.         Names.nextSurname(),  
5.         (BEGIN_ID + i) // BEGIN_ID = 135788  
6.     );  
7. }
```

Elke boogschutter die aangemaakt wordt in deze loop krijgt een derde id parameter toegewezen, wat gebaseerd is op het begin id (135788) en de index van de loop.

Boogschutter toString methode

Er werd gerefereerd naar een `toString()` methode, deze methode retourneert een String van informatie over de boogschutter in het volgende formaat:

"135787 (225/ 301) Nico Tromp".

In deze String zijn een aantal eigenschappen terug te vinden van een boogschutter:

- Het id
- De behaalde score
- De totale score
- Voornaam
- Achternaam

Het id eigenschap werd reeds bijgehouden bij een boogschutter en de behaalde score en totale score zijn berekeningen. Wat nog niet werd bijgehouden was de voornaam en achternaam van een boogschutter. Hier heb ik twee private String properties voor aangemaakt en gelijk twee getters voor geschreven. In de constructor parameters van een boogschutter stonden deze properties wel gedefinieerd, maar werden nog niet toegewezen aan de boogschutter. Dit heb ik opgelost door de properties te zetten in de constructor.

De behaalde score en totale score haal ik op met de aangepaste `getScore()` en `getTotalScore()` functies. Hoe ik deze heb geïmplementeerd lees je in de volgende opdracht "Voeg het score systeem toe".

Voeg het score systeem toe

Nu wordt de opdracht interessant. Er is niet specifiek vastgesteld hoe de structuur voor scores er uit komt te zien, dus ik ben met het volgende opgekomen (zie Figuur 2). Deze structuur bestaat uit een `int[][]` array, dit wordt ook wel een matrix array genoemd. De buitenste array staat gelijk aan de ronde en wordt dus geïnitieerd met een lengte gelijk

```
[
  [1, 8, 9]
],
[
  [2, 2, 4]
],
[
  [1, 7, 5]
],
enz..
```

Figuur 2: Ronde/score structuur

aan het maximaal aantal rondes. De binnenste array van 3 waardes staat gelijk aan de score die de boogschutter in deze ronde heeft behaald. Deze arrays worden geïnitieerd met de waarde van het maximaal aantal pijlen van de boogschutter.

Het aanmaken van de score array is simpel, dit maak ik een property van de boogschutter klas zodat ik hier overal bij kan binnen de boogschutter.

```
1. private int[][] scores;
```

Vervolgens initialiseer ik de array met de benodigde lengtes bij het aanmaken van een boogschutter.

```
1. private Archer()
2. {
3.     this.scores = new int[MAX_ROUNDS][MAX_ARROWS];
4. }
```

Ik heb voor deze datastructuur niet gekozen voor een ArrayList of andere vorm van java provided collectie klasse. Hiervoor heb ik gekozen omdat het aanmaken van al deze objecten overbodig is en deze opdracht makkelijk opgelost kan worden met een native array. Ook ga ik er niet van uit dat de datastructuur dynamisch is en heb ik daarom ook geen functies zoals `add()` en `remove()` nodig.

De functionaliteit van het berekenen van rondes en scores was al geleverd in het startproject, dus daar ga ik niet dieper op in. Wel heb ik twee functies aangepast:

```
1. public int getTotalScore()
2. {
3.     int total = 0;
4.
5.     for (int[] round : this.scores) {
6.         for (int score: round) {
7.             total += score;
8.         }
9.     }
10.
11.     return total;
12. }
```

De `getTotalScore()` functie retourneert de totale score van de boogschutter over alle rondes en pijlen. Dit wordt opgelost door twee simpele for-loops door de matrix array die alle scores bij elkaar optellen.

```
1. public int getMaxScore()
2. {
3.     return MAX_SCORE * MAX_ARROWS * MAX_ROUNDS;
4. }
```

De `getMaxScore()` functie retourneert het maximaal aantal punten wat de boogschutter kan verdienen over alle rondes en pijlen. Hier wordt gebruik gemaakt van de formule:

*maximaal aantal punten (10) * maximaal aantal pijlen (3) * maximaal aantal rondes (10)*

Wat in dit geval 300 terug geeft.

Vind de kampioen

Nu de score functionaliteit is gebouwd. Is het tijd om de boogschutters' scores te kunnen vergelijken met elkaar. Dit heb ik opgelost door Archer de Comparable interface te laten implementeren.

```
1. public class Archer implements Comparable<Archer>
```

Ik geef aan in de diamond operator dat de Archer class comparable is met een andere Archer.

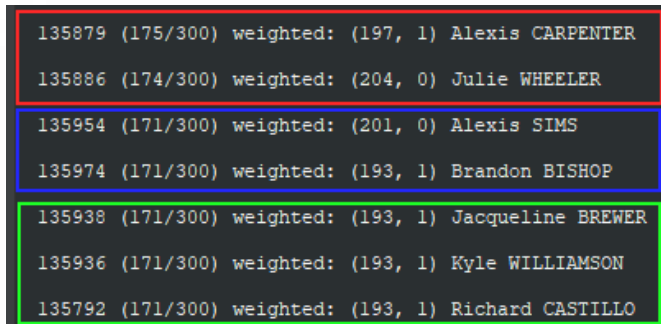
Vervolgens implementeer ik de `compareTo()` methode die met de Comparable interface komt als volgt:

```
1. @Override
2. public int compareTo( Archer archer )
3. {
4.     if (archer.getTotalScore() != this.getTotalScore()) {
5.         return archer.getTotalScore() - this.getTotalScore();
6.     }
7.
8.     if (archer.getWeightedScore() != this.getWeightedScore()) {
9.         Return archer.getWeightedScore() - this.getWeightedScore()
10.    }
11.
12.    return archer.getId() - this.getId();
13. }
```

De compareTo methode geeft altijd een integer terug, op basis van dit getal worden de twee boogschutters gesorteerd.

Eerst kijken we naar de 2 boogschutter scores. Als een positieve waarde wordt geretourneerd is de huidige boogschutters' score lager dan de andere boogschutters' score. Met een negatieve waarde is natuurlijk de boogschutters' score hoger. Mocht er precies 0 geretourneerd worden, dan zijn beide boogschutter scores gelijk. In dit geval komt er een extra conditie bij. Elke boogschutter heeft een "weighted score" dit is de achterliggende score dat rekening houdt met het aantal gemiste pijlen. Hoe meer pijlen een boogschutter mist (score van 0) hoe zwaarder deze score gestraft wordt. Deze berekening is te vinden in de functie `getWeightedScore()`. Om er achter te komen welke boogschutter de hoogste weighted score heeft doen we weer precies hetzelfde als de normale score. Mocht zelfs de

twee weighted scores gelijk zijn. Dan wordt de boogschutter met het hoogste id bovenaan gezet. Een voorbeeld van een gesorteerde ranking is te vinden in Figuur 3.



| | | | |
|--------|-----------|--------------------|-------------------|
| 135879 | (175/300) | weighted: (197, 1) | Alexis CARPENTER |
| 135886 | (174/300) | weighted: (204, 0) | Julie WHEELER |
| 135954 | (171/300) | weighted: (201, 0) | Alexis SIMS |
| 135974 | (171/300) | weighted: (193, 1) | Brandon BISHOP |
| 135938 | (171/300) | weighted: (193, 1) | Jacqueline BREWER |
| 135936 | (171/300) | weighted: (193, 1) | Kyle WILLIAMSON |
| 135792 | (171/300) | weighted: (193, 1) | Richard CASTILLO |

Figuur 3: Score systeem

Boogschutters in het rood zijn gerankt op basis van score (175 > 174).

Boogschutters in het blauw zijn gelijk geëindigd dus zijn gerankt op basis van weighted score (201 > 193)

Boogschutters in het groen zijn beoordeeld op id, omdat zowel de normale score als de weighted score hetzelfde waren.

Nu is de implementatie voor een comparable Archer afgerond en kunnen deze worden gesorteerd door de `sort()` methode aan te roepen van de `Collections` class en een lijst van archers mee te geven.

Efficiënt sorteren

Nu we eenmaal de lijst van boogschutters kunnen sorteren, gaan we kijken welk sorteer algoritme het meest efficiënt is voor het sorteren van het toernooi. Ik ga 3 sorteer algoritmes met elkaar vergelijken namelijk:

- Insertion sort
- Quick sort
- De standaard `Collection.sort()` methode

Elk sorteer algoritme wordt getest startend met 100 boogschutters tot en met 3.276.800 boogschutters. Voor een betrouwbaar resultaat run ik alle tests 10 keer en neem hiervan het gemiddelde. De tests zien er als volgt uit:

```
1. /**
2.  * Test sorting efficiency of standard Collections.sort sort.
3.  *
4.  * @param repetitionInfo repetitionInfo
5.  */
6. @RepeatedTest(17)
7. public void collectionSort( RepetitionInfo repetitionInfo )
8. {
9.     if (repetitionInfo.getCurrentRepetition() != 1) {
10.         int nrOfArchers = 100;
11.
12.         for (int i = 1; i < (repetitionInfo.getCurrentRepetition() - 1); i++) {
13.             nrOfArchers *= 2;
14.         }
15.
16.         ChampionSelector
17.             .collectionSort(
18.                 unsortedArchers.subList(0, nrOfArchers),
19.                 this.comparator
20.             )
21.         ;
22.     }
23. }
```

Ik maak gebruik van de `@RepeatedTest` decorator die standaard komt met het JUnit test framework. Met dit attribuut kan je aangeven hoe vaak een test opnieuw uitgevoerd mag worden. Met het `RepetitionInfo` object kan ik uitlezen in welke iteratie de test zich bevindt. Heel handig om zo het totaal aantal boogschutters uit te kunnen rekenen. Dit object heeft wel een iteratie nodig om geïnstantieerd te worden, dus daarom laat ik de eerste iteratie nog geen boogschutter lijst gesorteerd worden zodat dit niet meegenomen wordt in de resultaten. 17 iteraties van de test komt uiteindelijk op 3.276.800 boogschutters.

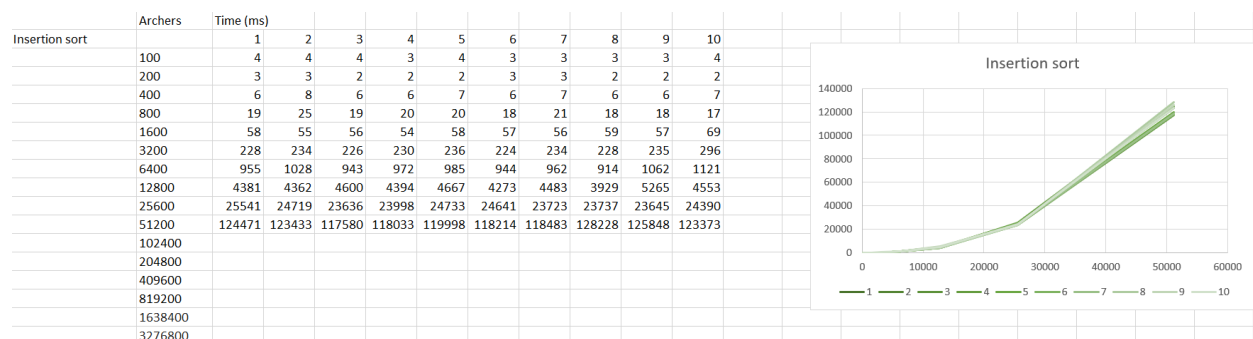
De tests zijn uitgevoerd met `-Xint` als JVM parameter. Dit zorgt ervoor dat het programma uitvoert in "interpreted-only mode". Performance benefits van de Java VM client adaptive compiler zijn niet aanwezig in deze modus.

Insertion sort

Dit algoritme is langzamer hoe meer elementen er gesorteerd worden, het is een kwadratische stijging van nature. Na 512000 elementen duurde het langer dan 20 seconden om de lijst te sorteren. Wel is dit algoritme zeer consistent, bleek uit mijn test. Er was weinig verschil te merken over de 10 verschillende tests die ik heb afgenomen. Dit algoritme is te gebruiken wanneer je middelmatig grote lijsten consistent zou willen sorteren.

Dit algoritme heb ik geïmplementeerd met gebruik van het boek Algorithms Fourth edition by Robert Sedgewick and Kevin Wayne page no. 245.

Big O notatie: $O(n^2)$

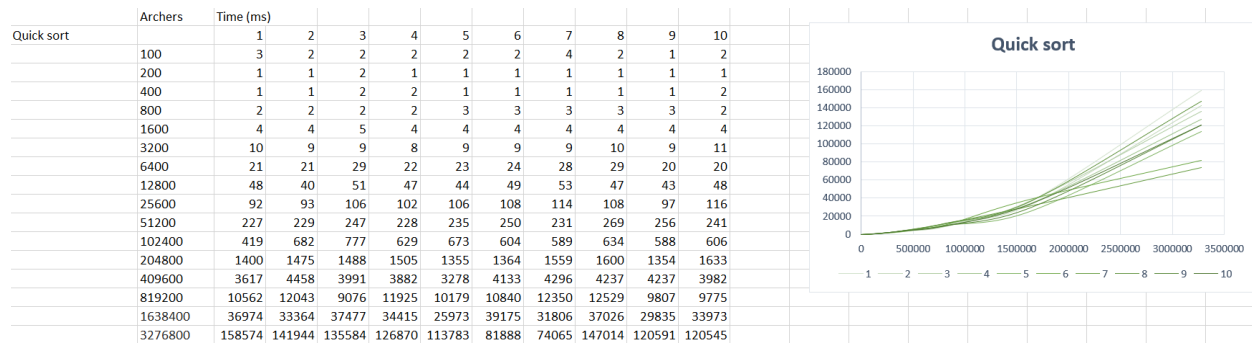


Quick sort

Dit algoritme is zeer inconsistent, maar kan het snelste zijn van de 3 algoritmes. Dit algoritme heb ik geïmplementeerd met gebruik van de volgende bron:

<https://www.geeksforgeeks.org/quick-sort/>

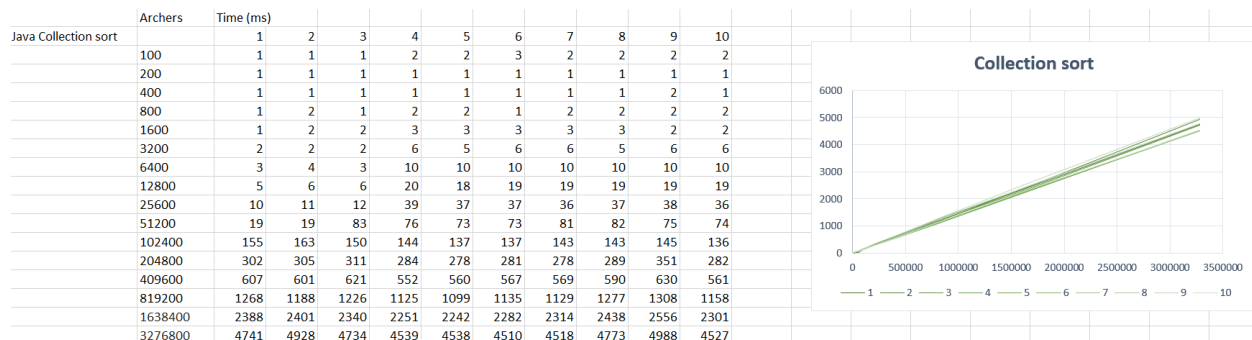
Big O notatie: $O(n \log n)$



Java collection sort

Door de dubbele for-loop lijkt dit op een kwadratische of exponentiele functie, maar dit is een lineaire functie.

Big O notatie: $O(n)$



Conclusie

Het is belangrijk om vooraf te bedenken wat voor sorteer algoritme je gaat gebruiken voor welke klus. Je kan met een aantal factoren rekening houden zoals snelheid, complexiteit van de datastructuur en de hoeveelheid data die je wilt sorteren. Een verkeerde keuze kan leiden tot zeer trage applicaties of zelfs crashes.