Algoritmer og datastruktur

Arbeidskrav 04

Gruppe 17: John I. Eriksen og Emil Slettbakk

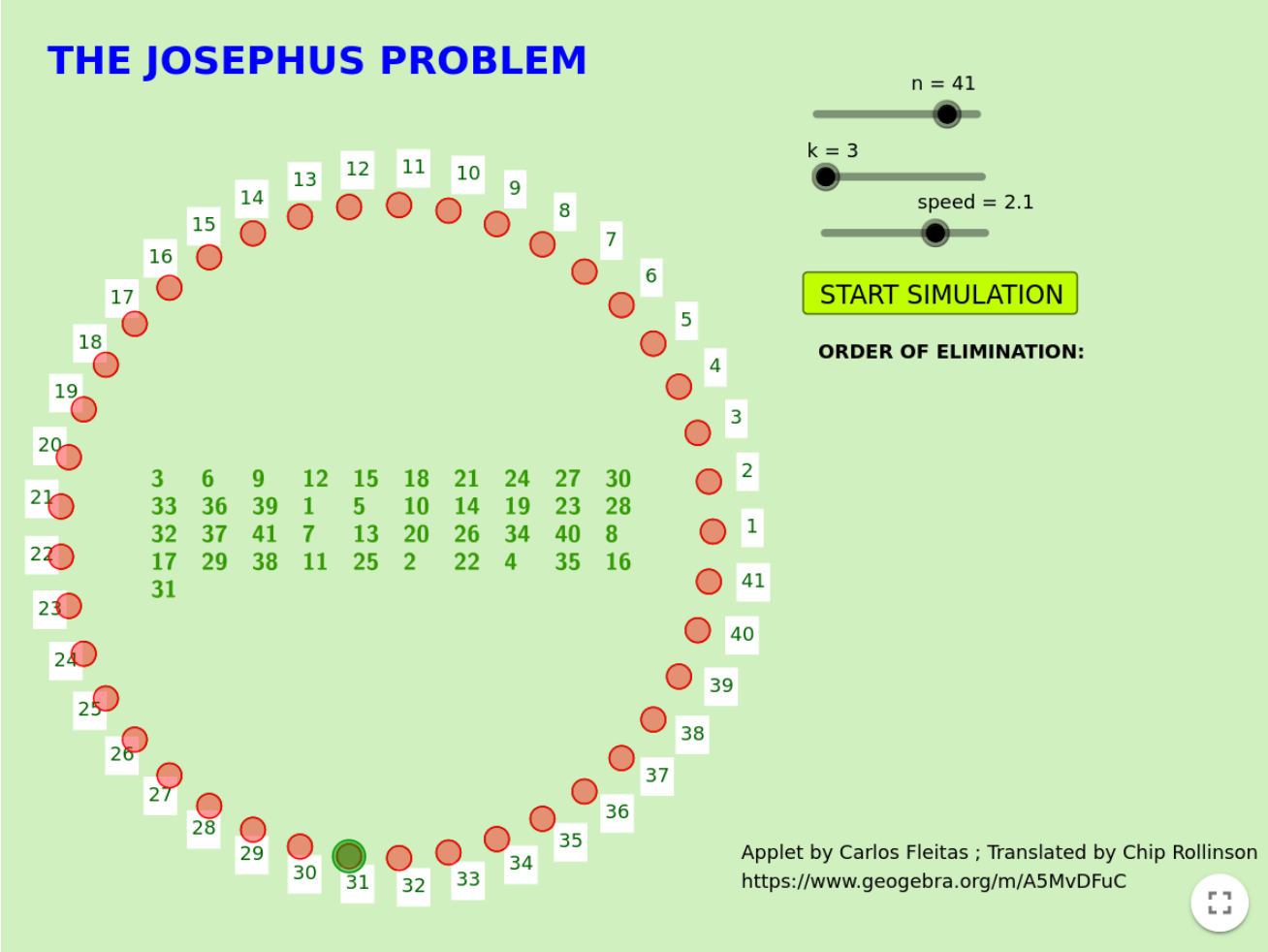
Oppgave 1: Josefus Flavius' problem

Programmet beregner hvilken posisjon Josefus Flavius må stille seg på, i en sirkel med n personer hvor de skal begå selvmord i rekkefølgen hver m-te person.

Hvis $m = 3$ betyr det at rekkefølgen starter slik:
[1] lever, [2] lever, [3] selvmord, [4] lever, [5] lever, [6] selvmord

For å sjekke at programmet kalkulerte riktig ble resultatet sammenlignet med en Josefus-kalkulator laget i Geogebra (https://www.geogebra.org/m/ExvvrBbR).

Skjermskuddet under viser kalkulatorens løsning på problemet med "historisk korrekt" tall $n = 41$ og $m = 3$:



En lite tabell med resultater fra kalkulatoren (som antas å være til å stole på) og programmet:

n	m	Posisjon, Java	Posisjon, kalkulator
11	7	5	5
16	4	1	1
25	19	15	15
41	**3**	**31**	**31**
50	5	19	19

Konklusjonen er at programmet utfører kalkulasjonen for optimal posisjon korrekt.

Tidskompleksitet

Tidskompleksiteten blir som følger:

$$n + n \cdot m$$
$$= n \cdot (m + 1)$$

Dette pga. to nøstede `while`-løkker en `for`-løkke.

Det ble observert to ulike atferder på tallene på de to ulike testene:

Test 1: Øker først $n + 1$ (antall personer) og så m (antall steg) over hele lengden til m.

```
1
2  // Incrementing full m for every incrementation of n
3  for (int i = 1; i < n + 1; i++) {
4      for (int j = 1; j < m + 1; j++) {
5      JosefusCircle(i, j);
6      }
7  }
```

Test 2: Øker kun m (antall steg), med antall personer konstant.

```
1  // Incrementing only m
2  for (int j = 1; j < m+1; j++) {
3      JosefusCircle(n, j);
4  }
```

Ved å skrive ut tiden på utvelgelsen ser man at i Test 2 bygger tiden seg opp rask til stegene er rundt 25, og deretter faller ned relativt mye før tiden igjen bygger seg opp. Test 2 ble kjørt ved $n = 1000$ personer (konstant) og $m = 200$ steg (itererer over hele intervallet).

Dette samme mønsteret kunne vi ikke finne igjen i Test 1, i alle fall ikke like markant, hvor tiden så ut til å øke mye mer konstant. Med ved å teste ved flere forskjellige verdier n og m kunne vi se at det var variasjon i tidsforbruk. Blant annet ville n og m satt lik 100 gi tidsforbruk som ca. doblet seg vekselvis. Dette var heller ikke konstant gjennom hele iterasjonssekvensen, så man kan konkludere med at verdiene av n og m påvirker tidsforbruket i kalkulasjonen.

Det ble ikke laget et plott for Test 2, siden disse inneholder en veldig stor serie datapunkter, men under er to utdrag fra output-en fra den, i tillegg til et av plottene fra Test 1.

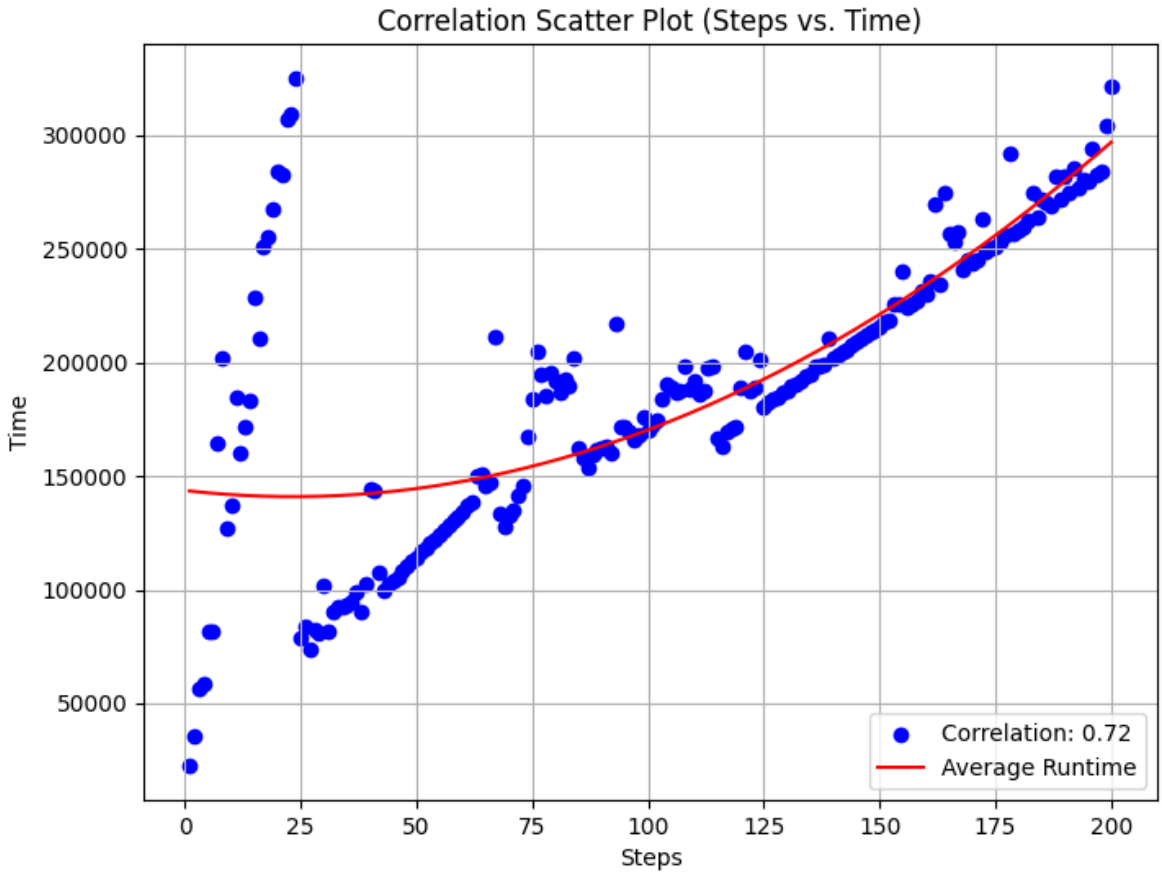
Utdrag 1 fra Test 1, $n = 100,\ m = 100$:

```
90, 17, 2280, 72
90, 18, 20021, 24
90, 19, 2630, 1
90, 20, 2710, 5
90, 21, 2900, 9
90, 22, 11260, 18

(...)

90, 34, 4631, 49
90, 35, 16860, 19
90, 36, 7300, 70
90, 37, 7390, 56
90, 38, 18840, 57
90, 39, 7310, 61
90, 40, 9990, 78
```

Utdrag 2 fra Test 1, $n = 100,\ m = 100$:

```
37, 18, 1020, 31
37, 19, 1050, 27
37, 20, 1130, 33
37, 21, 1361, 1
37, 22, 1230, 12
37, 23, 1500, 16
37, 24, 1370, 3
37, 25, 1370, 30
37, 26, 1480, 13
37, 27, 1480, 5
```

Plott fra Test 2, $n = 1000$ og $m = 200$:



Man ser tydelig en "drop-off" rundt 25 steg.

Oppgave 2: Matche parenteser, klammer og krøllparenteser

Programmet tar inn en gitt kildekode; for testing bruker vi Josefus-koden. Programmet går gjennom kildekoden og søker etter feil med parenteser, klammer eller krøllparenteser.

Dette gjøres ved å bruke en `stack`-implementasjon i Java. Hvis en linje i koden har en startparentes eller en startklamme, blir disse lagt til i stacken. Deretter sjekkes de for matchende sluttparentes eller klamme ved hjelp av metoden `isValidClosing`. Hvis programmet genererer en feilmelding, vil det peke brukeren til den linjen hvor feilen oppsto.