DAKI Programmeeropdracht 4: Breuken sorteren

(Deze versie is van 19 mei 2020, 00:37)

Je moet deze programmeeropdracht **zelf** en **alleen** maken. Je mag zeker met anderen overleggen over je aanpak, maar code van anderen bekijken of overnemen of zelf code delen met anderen is uitdrukkelijk niet toegestaan. Je moet je programma schrijven in C#, en inleveren via DOMjudge.

1 Opdrachtbeschrijving

DAKIBOT heeft simpelweg genóten van het werken met paren gehele getallen. Gehele getallen zijn sowieso geheel en al eerlijk, en oh zo integer. Ze wordt zich bewust van een sluimerend gevoel van... *begrip*... voor de uitspraak van de Duitse wiskundige Leopold Kronecker (1823–1891) "Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk".[1] Ze besluit nog wat meer met paren gehele getallen te doen, namelijk ze te beschouwen als de teller en noemer van een breuk, en die breuken te gaan sorteren in oplopende numerieke waarde. Daarbij wil ze natuurlijk in het domein van de gehele getallen blijven, en er geen vieze kommagetallen van maken en zich problemen door afrondingsfouten en beperkte nauwkeurigheid op de DAKInek te halen. Gelukkig kan dat.

Het is haar niet helemaal duidelijk waarom dit sorteren precies moet gebeuren. Iemand zei "omdat het kan", wat ze niet zo'n sterk argument vindt, maar ze heeft de 'desire' en de 'intention' om behulpzaam te zijn en 'belieft' dat ze het kan. Om er nog maar eens een quote van niveau-Kronecker tegenaan te gooien: "Ik heb het nog nooit gedaan, dus ik denk dat ik het wel kan" (niet van Pipi Langkous).

Jij gaat DAKIBOT helpen met een algoritme om breuken te sorteren. Je besluit vooral quicksort te gebruiken, omdat het zo... quick is, maar voor het sorteren van 'korte' segmenten implementeer je selection sort. Gelukkig staat selection sort beschreven in een opgave in CLRS. Een 'kort' segment is elk segment dat korter is dan of even lang is als een zekere drempelwaarde, die als onderdeel van de input is gegeven. Je moet selection sort gebruiken voor *alle* korte segmenten, dus ook de segmenten die je tegenkomt wanneer je quicksort recursief aanroept. Doe deze check dus niet alleen aan het begin van het sorteren van de gehele lijst, maar steeds als je een segment wil gaan sorteren: is het 'kort', gebruik dan selection sort, en anders quicksort.

Merk op dat deze opdracht **anders dan vorig jaar** is. Toen moest je quicksort combineren met een ander algoritme. Dus **je kunt code van vorig jaar niet hergebruiken**!

[1] "De gehele getallen zijn door de goede God gemaakt, al het andere is mensenwerk".

2 Invoer en Uitvoer

De invoer heeft op de eerste regel een getal n, maximaal $9\,999\,999$, en een getal k, maximaal $999\,999$. Dan volgen n regels met elk twee integers, beide maximaal $999\,999\,999$: de teller en noemer van een breuk. De noemer is positief. De betekenis van de parameter k wordt hieronder bij "Algoritmische eisen" uitgelegd.

De uitvoer heeft op de eerste regel weer het getal n, en op de regels erna dezelfde breuken als in de invoer, maar dan *oplopend gesorteerd op grootte van de waarde van de breuk*. De breuken worden niet vereenvoudigd (dus $\frac{4}{6}$ blijft $\frac{4}{6}$ en wordt niet $\frac{2}{3}$). Breuken met dezelfde waarde, zoals $\frac{7}{3}$ en $\frac{14}{6}$, moeten in oplopende volgorde van hun *noemer* staan, dus $\frac{7}{3}$ komt voor $\frac{14}{6}$ (want $3 < 6$) en $-\frac{223}{6}$ komt voor $-\frac{446}{12}$ (want $6 < 12$).

3 Voorbeeld

Dit voorbeeld laat zien dat gelijke waarden worden gesorteerd op noemer, want bij deze invoer

```
6 2
4 12
7 21
5 15
1 3
8 24
10 30
```

hoort de volgende uitvoer:

```
6
1 3
4 12
5 15
7 21
8 24
10 30
```

4 Algoritmische eisen

De opdracht heeft twee verschillende aspecten: het representeren van de objecten die je wil sorteren, en het sorteeralgoritme zelf. Het is aan te raden om te **proberen die twee onderdelen zoveel mogelijk onafhankelijk van elkaar te houden**, om zo de implementatie *begrijpelijk* te houden, en je sorteeralgoritme *herbruikbaar*. Je kunt dat doen door voor de 'grens' tussen het sorteeralgoritme enerzijds en de objecten die het sorteert anderzijds, een zogenaamde 'interface' te gebruiken. Die specificeert richting het sorteeralgoritme dat een object een bepaalde functionaliteit levert, in dit geval de mogelijkheid zichzelf met een ander object te vergelijken. Voor het *vergelijkingsgebaseerd* sorteren van objecten hoef je over die objecten zelf namelijk niets te weten, *behalve* hoe ze zich in de gewenste sorteervolgorde tot elkaar verhouden.

Representeren en vergelijken

Bij het vak *Modelleren en Programmeren* heb je kennis gemaakt met het begrip Interface. C# kent een interface `IComparable`, en de klasse die je schrijft om een breuk te representeren moet die interface implementeren. Als je dan je sorteeralgoritme zegt dat het als input objecten krijgt die die interface implementeren, kan je algoritme objecten met elkaar vergelijken (want dat wordt gegarandeerd door de Interface), zonder dat het hoeft te weten wat voor objecten het zijn, en hoe die het onderling vergelijking implementeren.

Je implementeert de interface door in de `Breuk` klasse een '`compareTo`' methode te schrijven, die zichzelf met een andere breuk kan vergelijken, en een geheel getal (≤ 0, 0, of ≥ 0) teruggeeft (zie het C# manual hierover). Je mag in dit verband breuken **niet vergelijken** door bij twee breuken de teller te delen door de noemer, en de waardes met elkaar te vergelijken. Door afrondfouten kan de uitkomst dan namelijk onjuist zijn. Je moet een *exacte berekening* maken die van de tellers en noemers gebruik maakt, en alléén gehele getallen gebruikt.

Sorteren

Je moet voor het sorteren een hybride van quicksort en selection sort implementeren: Voor *álle* lijsten van lengte $> k$ (waar k gegeven is als het tweede getal op de eerste regel van de invoer) moet je programma quicksort gebruiken, en voor lijsten van lengte $\leq k$ moet het selection sort gebruiken. Je moet beide algoritmen zelf implementeren. Let op: het woord "*álle*" hierboven betekent dat deze eis in *elke* (recursieve) aanroep van quicksort opnieuw geldt: een segment met $\leq k$ elementen wordt met selection sort gesorteerd, en bij een segmentlengte $> k$ moet je partitioneren en tweemaal in recursie gaan (waarna je dus voor elk van beide partities opnieuw moet beslissen of je quicksort of selection sort moet gebruiken).

N.B.: Een dergelijke hybride vorm van twee sorteeralgoritmen is vaak sneller dan één van beiden; quicksort werkt goed voor grote lijsten maar is minder efficiënt op kleinere lijsten. De waarde van k is niet van invloed op de uitkomst, maar *wel* op de verwerkingstijd! Je kunt voor jezelf bekijken bij welke waarde van k je programma het snelst werkt.

5 Hints, Tips

Geen, behalve wat hierboven allemaal staat.