Het 'arglistige inspectie' algoritme

Daniel von Asmuth

Samenvatting

Een poging om een programmeeropgave van Johan Volkers op te lossen.

De opgave

Johan Volkers introduceerde de volgende programmeeropgave bij de InteresseGroep Programmeren van de HCC:

Schrijf een programma in een willekeurige taal dat het volgende doet

- Input is een file met iets minder dan 1.000.000 regels.
- Iedere regel bevat een uniek getal tussen de 1 .. 1.000.000 in een willekeurige volgorde
- Geef aan welke getallen **niet** in de file staan.

Beperkingen

- het programma mag hoogstens één array van integers gebruiken met een maximale grootte van tien elementen.
- Er mag geen externe sort gebruikt worden.

Bij de opgave behoort ook een voorbeeldbestand. Dat blijkt getallen van 0 t/m 1.000.000 te bevatten met 1 getal per regel en 80 getallen in dat bereik ontbreken. Ik nam de vrijheid om de opgave zo te veranderen dat het gaat om getallen van 0 t/m 999.999.

De opgave zou zijn ontleend aan 'Programming Pearls' van Jon Bentley. Wat de juiste oplossing zou zijn is niet helemaal duidelijk geworden. We gaan ervan uit de uitdrukking 'externe sort' verwijst naar een routine die de getallen op volgorde sorteert, gebruikmakend van bestanden op een harde schijf of tape, in tegenstelling tot sorteren in het interne geheugen. Eigenlijk zou de opgave gewijzigd moeten worden in een invoerbestand met (bijna) een biljoen getallen.

1 Een mogelijke oplossing

De opgave is vrij eenvoudig geformuleerd en op te lossen: lees alle getallen in het invoerbestand sequentieel en test ieder getal of het gelijk is aan 1, daarna lees je het hele bestand door om te testen of de 2 erin voorkomt, tot en met 1000000. De getallen die niet voorkomen druk je af.

2 Een betere oplossing

De vorige oplossing is correct, maar erg langzaam. Snelle algoritmen zijn betere algoritmen.

Maak een bestand aan van 1 miljoen bits (of bytes) en vul het met nullen (in Unix hoef je enkel een leeg bestand aan te maken). Daarna lees je het invoerbestand regel voor regel in. Als je het getal g aantreft schrijf je een 1 naar het g-e bit of byte in het hulpbestand. Als je klaar bent lees je het hulpbestand bit voor bit en als je een 0 aantreft, druk je het volgnummer van dat bit af.

Dit is uiteraard sneller. Een nadeel is dat magnetische harde schijven lees-schrijfkoppen hebben die telkens een spoor moeten zoeken om data te lezen of schrijven. Tegenwoordig is de zoektijd in de orde van 0,1 milliseconde en moet je ongeveer 2 ms wachten voordat de gewenste sector is bereikt. Nu moet de schijf telkens verspringen tussen het invoer- en uitvoerbestand, dus 4 ms per getal ofwel ruim een uur geschatte verwerkingstijd. Door *disc caching* zal de echte snelheid een stuk hoger uitpakken.

Het is sneller om bestanden sequentieel te lezen en schrijven; voor tape is dat de enige mogelijkheid. Voor halfgeleidergeheugens zoals SSDs geldt dit niet en hebben we een goede oplossing.

3 Een goede oplossing: het QuickCheck algoritme

Johan Volkers en ik kwamen met de volgende oplossing, die vervolgens geaccepteerd werd. Ik bedacht de naam 'QuickCheck' omdat ze veel lijkt op het QuickSort algoritme van C.A.R. Hoare; we moeten ons dus afvragen of we hier een externe sortering toepassen.

Schrijf een recursieve functie die als parameters de naam van het te verwerken bestand krijgt en de minimale en maximale getallen die erin kunnen voorkomen, bijvoorbeeld 1 en 1000000. Je bepaalt de pivot $p = (min + max)/2$. Je maakt twee hulpbestanden A en B. Getallen die kleiner of gelijk zijn aan de pivot schrijf je naar A, grotere naar B en je telt hoeveel getallen je naar A en B hebt geschreven. Als het aantal getallen in een bestand gelijk is aan het aantal getallen in de betreffende reeks weet je dat het bestand geen van de gevraagde getallen heeft en kun je het verwijderen. Anders moet je het bestand recursief doorzoeken en vervolgens verwijderen.

Intermezzo

Sorteren kan duiden op artikelen in groepen verdelen of dingen op een bepaalde volgorde leggen. Informatici gebruiken meestal de laatste betekenis. Het QuickCheck algoritme sorteert in de eerstgenoemde zin van het woord. Het verdeelt een bestand in tweeën op basis van het meest significante bit. Het Arglistige Inspectie algoritme gebruikt daarvoor willekeurige bitposities.

Hoe zou je de mogelijke getallen verder kunnen classificeren? Een mogelijkheid is om de verzameling van cijfers te gebruiken die in het getal voorkomen. Voor c cijfers en p posities blijven er van de c^p getallen dan $\binom{c}{n} = \frac{c!}{n!(c-n)!}$ klassen over met n verschillende cijfers. De som voor $n =$ 1 t/m 6 bedraagt dan 847 met c = 10 cijfers.

4 Het Arglistige Inspectie algoritme (binair)

Is er een snellere oplossing mogelijk? Het idee was dat het QuickCheck algoritme 20 recursieniveaus nodig heeft om 1 bestand met 1 miljoen getallen te splitsen in 1 miljoen bestanden met 1 getal, zodat het programma in totaal 20 miljoen getallen moest testen: we kunnen dat proberen te verbeteren door de invoer in meer dan 2 delen op te splitsen.

Het volgende algoritme is genoemd naar het personage Inspecteur Arglistig uit de jongensboeken van Wim van Helden. Het overtreedt de regels door een array van 20 gehele getallen te gebruiken, 1 per bit positie, waarmee getallen van 0 t/m 2^{20} geteld kunnen worden.

De verwerking wordt versneld door invoergetallen op te splitsen in 2^5 tijdelijke bestanden, zodat de recursiediepte beperkt blijft tot 4 niveaus en de bestanden kleiner zijn.

Elk bestand wordt nu in twee doorgangen verwerkt: eerst worden alle getallen gelezen om te tellen op welke bit posities een 1 staat. De vijf bits die het minst vaak geteld zijn worden gebruikt om de getallen het bestand in de tweede doorgang te verdelen over 32 hulpbestanden. De hulpbestanden zijn in binair formaat, zodat het aantal getallen volgt uit de grootte van het bestand. Het programma test welke bestanden de maximale grootte hebben bereikt, verwijdert die en verwerkt het restant recursief. Om de code te vereenvoudigen worden de getallen 1000001 t/m 1048575 toegevoegd aan de hulpbestanden, terwijl ze in het origineel niet voorkomen.

De recursie eindigt als een bestand leeg is. Het programma drukt dan het getal af dat erin gestaan zou hebben. Het is echter mogelijk dat er meerdere getallen ontbreken in hetzelfde bestand.

5 Arglistige Inspectie algoritme (decimaal)

De volgende variant houdt zich beter aan de regel en gebruikt een array van 10 gehele getallen. Daarmee kunnen we tellen hoe vaak elk cijfer in de invoer optreedt. Afgezien van het getal 1,000,000 kan bevatten bevat het testbestand getallen van (maximaal) 6 cijfers.

Na het tellen van die cijfers wordt het minst voorkomende cijfer c gekozen en in de tweede doorgang wordt de invoer verdeeld over maximaal 7 hulpbestanden aan hand van dat cijfer: getallen met c in de eerste positie worden naar het eerste bestand geschreven, enzovoorts. Getallen die het cijfer c niet bevatten gaan naar het laatste bestand. Als dat cijfer er vaker in voorkomt, belandt het getal ook in meerdere bestanden. Het gevolg is dat de gebruiker de uitvoer moet sorteren en duplicaten verwijderen.

Als het invoerbestand c cijfers heeft op p posities, dan zou het c^p getallen moeten bevatten indien er geen ontbrak. Voor de meeste hulpbestanden is dat c cijfers op p-*1* posities; voor het laatste zijn dat c-*1* cijfers op p posities.

Uitvoerbestanden die het maximale aantal getallen bevatten worden meteen verwijderd, de rest wordt recursief verder geïnspecteerd. Voor de bestanden met het gekozen cijfer blijven er nog p-*1* posities over om te doorzoeken, voor het andere nog c-*1* cijfers. De functie *inspectFile()* heeft parameters die aangeven welke cijfers en posities getest moeten worden; die verzamelingen zijn gecodeerd als enkele integers.

Als die functie een leeg bestand meekrijgt, dan moeten de gezochte getallen alle ontbreken – in ons geval gaat het steeds om 1 getal, dat dan wordt bepaald aan hand van de minimum waarde en de gezochte cijfers en posities en afgedrukt. Na de recursie worden overblijvende tijdelijke bestanden verwijderd.

6 Optimalisatie

Uiteindelijk publiceerde Johan ook zijn code. Bij bestudering blijkt dat hij voor tijdelijke bestanden met een bereik van 10 of minder getallen een array van 10 booleans gebruikt en dan de invoer verwerkt volgens de methode uit paragraaf 2. Mijn geval is iets complexer, maar ook hier waren significant minder tijdelijke bestanden nodig en werd het programma ook sneller.

7 Resultaten en discussie

Johans QuickCheck algoritme bereikte verwerkingstijden onder 3 seconden, waarin het 7.823.408 getallen las en 1748 tijdelijke bestanden maakte, terwijl 954 files worden verwerkt; hier gebruikte zijn Perl programma 6 seconden; herschrijven in Java reduceerde dat tot 3 seconden. Het resultaat is dat uit de reeks 0 t/m 10,0000 er 80 getallen ontbreken.

Het Arglistige Inspectie algoritme in Java produceert 6691 tijdelijke bestanden en heeft een looptijd van circa 5 seconden om 9.571.040 getallen te lezen. De verbeterde versie produceert slechts 4173 tijdelijke bestanden en leest 9.563.573 getallen in 4 seconden uit 1392 bestanden; twee derde deel wordt direkt verwijderd: Johans algoritme wist daarmee minder bestanden. De ironie is dat je externe sortering nodig hebt om de uitvoer op te schonen.

Het binaire algoritme in C heeft slechts 2 seconden nodig en maakt 5888 hulpbestanden aan; samen met het originele bestand leest het 3.993.090 getallen.

De verklaring van de resultaten bestaat uit het feit dat beide programma's per ronde de invoer splitsen en bestanden met maximale grootte overslaan, zodat er telkens 80 bestanden overblijven voor nadere geïnspectie—dat geldt niet voor de eerste paar rondes en voor de laatste.

Het QuickCheck algoritme zal de eerste 6 ronden 126 bestanden schrijven en 6.000.000 getallen verwerken. Dan volgen ongeveer 10 ronden waarin 160 bestanden worden geschreven = 1600 tijdelijke bestanden, waarvan de helft wordt overgeslagen en de laatste ronde wordt alleen gelezen.

Het aantal getallen per file daalt per ronde met een factor 2. We kunnen dus het aantal getallen ruw schatten op $80 \times (12500 + 6250 + 3125 + 1562 + 781 + 390 + 195 + 97 + 48 + 24 + 12) + 6.000.000 = 7998720$.

Het decimale algoritme is moeilijker in te schatten. Per ronde worden 7 tijdelijke bestanden geschreven. De eerste twee ronden levert dat 56 bestanden op en worden 4.000.000 getallen gelezen. Er blijven dan 4117 bestanden over, dus ongeveer 7 rondes waarin telkens 560 bestanden worden geschreven.

In de laatste ronde worden er enkel $80 \times 9 = 720$ getallen gelezen. Dan resteert 5562853 / 2 getallen voor het middendeel waarin 1304 bestanden worden gelezen die dus gemiddeld 2132 * 7 of 15681 getallen bevatten. Dat is vergelijkbaar met 12500 + 4166 + 1388 + 462 + 154 + 51 + 17 = 18738 als per ronde het aantal getallen per file met twee derde afneemt.

Conclusie

Het Arglistige Inspectie algoritme is in het voordeel totdat het aantal bestanden de maximale waarde heeft bereikt en daarna in het nadeel omdat het alle getallen twee keer leest. Dit nadeel kan vervallen door de cijfers in een vaste volgorde te verwerken, maar veel zal het niet opleveren.

Mijn code heeft een voordeel boven die van Johan: de C code maakt tijdelijke bestanden met binaire getallen: die zijn kleiner en sneller te verwerken; de Java code maakt ASCII bestanden, maar verwerkt de cijfers zonder conversie naar binaire vorm en terug.

Bronnen

https://nl.wikipedia.org/wiki/Inspecteur_Arglistig