Algoritmen en Datastructuren
Taak 3: Het partitieprobleem

Tom Naessens
Academiejaar 2010 – 2011

2 mei 2011

Algoritme 1: Recursief backtracking-algoritme

Werking algoritme

Het algemeen idee van het recursief backtracking-algoritme baseert zich erop dat het verschil tussen de som van de elementen van de grootste deelrij en van de kleinste deelrij minimaal moet zijn.
In de hoofdmethode `solve` van het algoritme worden enkele array's en integers geïnitialiseerd. De array `sums` wordt gevuldt met de som van elke partitie en het eerste minimale verschil wordt berekent via de methode `getVerschil`. Deze methode overloopt de array `sums` en haalt hier uit de kleinste en de grootste waarde van alle partities en geeft het verschil hiertussen terug. Hierna wordt de recursieve methode aangeroepen.
Deze recursieve methode `recursive()` berekent alle mogelijke waarden waarbij het eerste schotje op een gegeven plaats staat. We plaatsen de overige schotjes rechts van het eerste schotje en verschuiven deze. Na elke verschuiving berekenen we het verschil. Als bij een gegeven configuratie het verschil kleiner is dan het minimaal verschil, zetten we het minimaal verschil gelijk aan het nieuwe, kleinere, verschil. Anders, als het verschil groter is dan het optimaal verschil stoppen we met het verplaatsen van de schotjes en roepen we de functie opnieuw aan maar plaatsen we het eerste schotje 1 naar rechts zodat we toch niet alle mogelijkheden moeten overlopen voor we aan onze oplossing komen.
Als we uiteindelijk klaar zijn met deze methode gaan we terug naar onze hoofdmethode `solve` waar we de optimale configuratie teruggeven.

Optimalisaties

Ik heb in dit algoritme 1 optimalisatie doorgevoerd die toch een behoorlijke impact zal hebben op de uitvoeringstijd. Ik heb namelijk de methode `getVerschil` aangepast door enerzijds de methode zelf aan te passen en anderzijds de sommen van de elementen van alle partities op te slaan in een array, `sums` genaamd.
Voor deze optimalisatie werd doorgevoerd bestond de methode `getVerschil` uit een lus die alle partities, aan de hand van een array, de configuratie van de schotjes overliep, en daarin nog een lus om alle getallen binnen een partitie op te tellen. Dit zou meer dan lineaire tijd kosten.
Ik heb deze tijd gereduceerd naar een lineaire tijd door een array toe te voegen `sums` die de som van de elementen van de verschillende deelrijen bij houdt. Iedere keer er een schotje wordt verplaatst wordt de som in deze array aangepast wat in constante tijd gebeurd. Zo kunnen we ook onze methode `getVerschil` herschrijven zodat hij maar 1 lus gebruikt en in lineaire tijd

werkt.
Deze optimalisatie bracht wel mee dat ik een aantal keer enkele ingewikkelde bewerkingen heb moeten doen om de juiste getallen bij de juiste partitie op te tellen of af te trekken.
Door deze optimalisatie heb ik ook 2 verschillende array's moeten bij houden om in verschillende gevallen de som anders op te slaan. Als de schotjes herplaatst worden, meerbepaald als het eerste schotje opschuift, kan het nieuwe verschil tussen de som van de elementen van de deelrijen niet meteen berekent worden uit de array die gebruikt wordt bij het verplaatsen van de andere schotjes maar wel op basis van de array die in het begin gebruikt werd om het eerste verschil te berekenen. Als ik deze extra array niet had ingevoerd had ik toch 1 keer per recursieve oproep de oude, meer tijdrovende, methode `getVerschil()` moeten gebruiken. Nu is dat niet meer nodig en is deze manier dus sneller.

Algoritme 2: Gretig algoritme

Werking algoritme

Dit algoritme is relatief simpel ten opzichte van het recursief backtracking algoritme. Het is gebaseerd op het volgende:
We berekenen eerst het gemiddelde van de elementen van de hele array. Daarna lopen we door de array kijken we telkens als we een waarde tegenkomen waardoor de som over het gemiddelde zou gaan. Als dit niet het geval is gaan we verder met het volgende getal. Anders, als de waarde het gemiddelde zou overschrijden plaatsen we een schotje en berekenen we opnieuw het gemiddelde van de deelrij die nog niet verdeeld is. Hierna overlopen opnieuw we de waarden van deze rij totdat het gemiddelde weer overschreden is. We gaan zo door totdat de hele rij doorlopen is en uiteindelijk geven we een array met een configuratie van schotjes terug.

Optimalisaties

Buiten het netter schrijven van de code en de functie `Arrays.copyOfRange()` te gebruiken in plaats van handmatig in een lus een deelrij op te stellen, heb ik hier geen optimalisaties doorgevoerd.

Resultaten

Een voorbeeld van een rij die niet optimaal verdeeld wordt door mijn gretig algoritme is de volgende rij:

$$\{100, 200, 200, 300\},\ k = 3$$

Deze rij wordt door mijn algoritme verdeeld in

100|200 200|300

terwijl

100 200|200|300

een betere oplossing zou zijn.

Mijn algoritme komt aan deze suboptimale oplossing omdat hij eerst het gemiddelde van de hele rij berekent. De som van deze getallen is gelijk aan 800, dus het gemiddelde is 266. Daarna overloopt hij de rij. 100 is het eerste element en bevindt zich automatisch in de eerste partitie. Nu kijkt mijn algoritme als hij het tweede element, namelijk 200 er nog aan kan toevoegen zonder dat het gemiddelde overschreden wordt. Dit kan niet, dus plaats hij een schot en voegt hij 200 toe na dit schot. Hierna berekent hij het gemiddelde automatisch opnieuw maar dit keer enkel met de overgebleven elementen, namelijk 200 en 300. Het gemiddelde hier van is 250. 200 is kleiner dan 250 dus mag het nog bij de tweede partitie. Hierna wordt het gemiddelde overschreven doordat 300 wordt toegevoegd en wordt 300 uiteindelijk een schot geplaatst tussen de tweede 200 en 300 waardoor we als uiteindelijke oplossing 100|200 200|300 bekomen.