

Hertentamen Algoritmen en Programma's (I00008)

dinsdag 20 februari 2007, 13:30 - 15:30 uur, HG00.023

- Deze herkansing is gesloten boek. Je mag geen literatuur of PC gebruiken.
- Dit tentamen bestaat uit twee opgaven, die beide even zwaar meetellen voor het cijfer. De waardering per onderdeel staat vermeld in de kantlijn.
- Ga niet overhaast te werk en lees de opgaven nauwkeurig. Begin met de opgaven die je het eenvoudigst lijken. Denk eerst goed na. Schroom niet vragen te stellen over eventuele onduidelijkheden. Hou je antwoorden kort en controleer ze voor je het werk inlevert.

Succes!

1 Sorteren: Selectionsort en Quicksort

In deze opgave ga je proberen om het alom bekende *Quicksort* algoritme te versnellen. Als voorbereiding hierop kijken we echter eerst naar het niet veel minder populaire *Selectionsort* algoritme. Om te voorkomen dat je sorteermethodes door elkaar gaat halen geven we een korte beschrijving van beide algoritmes. Bij Selectionsort is er steeds sprake van een gesorteerd en een ongesorteerd deel. Het idee is om tijdens iedere sorteerslag het minimale element uit het ongesorteerde deel te bepalen en dit vervolgens achter aan het gesorteerde deel toe te voegen.

Quicksort is een recursieve, zogenaamde *divide-and-conquer* methode. Het idee is om allereerst één van de elementen in de te sorteren rij als *mediaan* aan te wijzen en vervolgens de rij in drie delen te splitsen: het eerste deel bevat de elementen kleiner dan de mediaan, het tweede deel elementen gelijk aan de mediaan en het derde deel de resterende elementen, die uiteraard groter dan de mediaan zijn. Een elegant algoritme voor het splitsen is het door Dijkstra bedachte D(utch)N(ational)F(lag)-algoritme. Het rode deel bevat na uitvoering van het DNF-algoritme alle elementen kleiner dan de mediaan, het witte gebied alle elementen gelijk aan de mediaan, en alle elementen groter dan de mediaan zitten in het blauwe deel. Na splitsing worden het eerste (rode) en het laatste (blauwe) deel recursief verder gesorteerd.

We beginnen met het verbeteren selection sort. Het idee is als volgt. Tijdens het doorlopen van het ongesorteerde deel op zoek naar het minimale element is er steeds sprake van een tot dan toe gevonden minimale element. Je dient er nu voor te zorgen dat dit *huidige minimale element* meteen, dus zodra je het bent tegengekomen, aan het begin van het ongesorteerde gebied gezet wordt. Bovendien moet je, ieder ander element uit de rij dat gelijk is aan dit element ook naar voren verplaatsen. Wat je hiermee bereikt is dat je tijdens iedere sorteerslag *alle* minimale elementen uit het ongesorteerde gebied naar voren hebt verplaatst.

Stel je hebt de gedeeltelijk gesorteerde rij:

0, 0, 1, 7, 5, 2, 3, 2, 9

waarbij het ongesorteerde deel bij het element met waarde 7 begint. Na één slag ziet de rij er dan als volgt uit:

0, 0, 1, 2, 2, 5, 3, 7, 9

Het gesorteerde deel is dan met twee elementen gegroeid.

5 punten

1.a) Implementeer dit algoritme in C++.

2 punten 1.b) Wat is de complexiteit van dit algoritme? Maak onderscheid tussen *best*, *average* en *worst* case complexiteit. Motiveer je antwoord duidelijk en beknopt.

En nu terug naar Quicksort.

1 punten 1.c) Wat is de complexiteit van Quicksort? Maak wederom onderscheid tussen *best*, *average* en *worst* case complexiteit. Ook hier weer een korte motivatie.

De verbetering die we aanbrengen is eenvoudig: zodra het nog te sorteren gebied minder dan ondergrens elementen bevat sorteren we niet recursief verder maar gebruiken het verbeterde selectionsort algoritme. Hier is *ondergrens* een of andere van te voren vastgestelde constante.

4 punten 1.d) Implementeer deze verbeterde Quicksort in C++. Hierin mag je gebruik maken van de procedure genaamd DNF, waarmee de rij in 3 delen gesplitst kan worden. De header hiervan is:

```
void DNF (int r [], int van, int tot, int med, int &wit, int &bla)
```

Iets preciezer: DNF splitst van de meegegeven rij *r* de elementen beginnende bij *van* tot aan *tot* in drie delen, waarbij *med* als mediaan gebruikt wordt. Na afloop eindigt het eerste (rode) deel bij *wit* en begint het derde (blauwe) deel bij *bla*.

2 punten 1.e) Wat is het effect van de verbetering op de complexiteit?

1 punten 1.f) Tot slot nog een andere vraag over Quicksort. Waarom mag Quicksort eigenlijk strict genomen geen *in-situ* methode genoemd worden?

2 Backtracking: de kroegentocht

In deze carnavalstijd is het niet ongebruikelijk om op een dag precies 11 kroegen te bezoeken en in ieder van die kroegen een of meer biertjes te drinken. We vragen je een algoritme te schrijven dat het snelste pad in daadwerkelijk reistijd bepaalt voor iemand die in de nabijheid van een aantal kroegen woont. Bij de kroegentocht dien je de volgende regels in acht te nemen:

- In totaal dienen precies 11 kroegen bezocht te worden.
- Men mag een kroeg voorbij lopen zonder binnen te gaan.
- De volgende kroeg die men bezoekt is niet niet gelijk aan één van de laatste drie kroegen die bezocht zijn (dit is inclusief de huidige kroeg). Kroegen mogen dus meer dan eens bezocht worden, als er maar tenminste 3 andere kroegen tussendoor bezocht worden.
- Het is niet noodzakelijk dat alle kroegen uit de tabel bezocht worden.
- Naarmate er meer kroegen bezocht zijn (en dus meer bier genuttigd is) neemt de daadwerkelijke reistijd toe ten opzichte van de reistijd in nuchtere toestand. Deze toename van de reistijd is in procenten gelijk aan het kwadraat van van het aantal bezochte kroegen. Na 5 kroegen neemt de reistijd dus met $5 \times 5 = 25\%$ toe ten opzichte de reistijd in volkomen nuchtere toestand. Na 10 kroegen is de benodigde reistijd dus het dubbele van wat er in onderstaande reistijdentabel staat (de toename is $10 \times 10 = 100\%$).
- De kroegentocht begint en eindigt thuis. De looptijden naar de eerste kroeg en van de laatste kroeg tellen mee in de totale tijd.

De afstanden van *i* naar *j* uitgedrukt in minuten lopen in nuchtere toestand is af te lezen uit een reistijdentabel. Op plek 0 staat het huis van onze kroegenloper. Voor een situatie met 7 kroegen is de tabel bijvoorbeeld.

```

const int AantalKroegen = 7;
int afstand[AantalKroegen+1][AantalKroegen+1]
= {{ 0, 9,11,15, 9, 8, 6, 5} // reistijd vanaf huis
  ,{ 9, 0, 2, 6,12,17,15,14} // reistijd vanaf kroeg 1
  ,{11, 2, 0, 8,14,16,14,13} // reistijd vanaf kroeg 2
  ,{15, 6, 8, 0, 6,11,15,20} // ..
  ,{ 9,12,14, 6, 0, 5, 9,14}
  ,{ 8,17,16,11, 5, 0, 4, 9}
  ,{ 6,15,14,15, 9, 4, 0, 5}
  ,{ 5,14,13,20,14, 9, 5, 0}
};

```

Een mogelijke kortste tocht is hier:

```

Lengte van de tocht = 88.05 minuten.
Naar 7 in 5 minuten.
Naar 6 in 5.05 minuten.
Naar 5 in 4.16 minuten.
Naar 4 in 5.45 minuten.
Naar 1 in 13.92 minuten.
Naar 2 in 2.5 minuten.
Naar 3 in 10.88 minuten.
Naar 4 in 8.94 minuten.
Naar 5 in 8.2 minuten.
Naar 6 in 7.24 minuten.
Naar 7 in 10 minuten.
Naar 0 in 11.05 minuten.

```

- 10 punten **2.a)** Implementeer in C++ een functie die via een backtrackalgoritme dat de snelste tocht langs 11 kroegen onder de boven gegeven condities bepaalt. De lengte van deze tocht wordt gemeten in de daadwerkelijke reistijd. Natuurlijk werkt je programma niet alleen voor bovenstaande tabel, maar voor iedere inhoud van zo'n tabel met minstens 4 kroegen.
- Hint:** Voor de snelste tocht tot nu toe en de lengte van die tocht mag je globale variabelen gebruiken.
- 3 punten **2.b)** Geef een programmafragment dat de snelste kroegentocht afdruckt zoals in het voorbeeld gegeven.
- 2 punten **2.c)** Beschrijf hoe het algoritme uit het eerste onderdeel geoptimaliseerd kan worden tot een *branch-and-bound* algoritme. Je hoeft niet de hele code opnieuw te geven, je kunt volstaan met een beschrijving van de benodigde wijziging(en).