Eerste deeltoets Algoritmiek (INFOAL)
01-10-2009
Uitwerkingen

(De opmaak is over de jaren verloren gegaan maar de antwoorden zijn nog correct en gecontroleerd door de docent)

Totaal 20 punten te verdienen, waarvan 4 voor de opgaven 1, 3 en 4. Opgave 2 over de Lift was 5pt waard en opgave 5 over DFS-nesting 3pt.

Opgave 1. Zoeken naar dubbele.
Dat er een dubbele is volgt uit de EIGENSCHAP, dat de LENGTE van het array (n-1+1) groter is dan het BEREIK van waarden (A[n]-A[1]+1). Het ligt dus het meest voor de hand om die eigenschap als invariant te houden, en hem voor steeds kortere segmenten te verkrijgen. De eigenschap geldt voor tenminste een helft van een segment weer, zodat je binair kunt zoeken:

```
i=1; j=n
// INV: A[j]-A[i] < j-i
while (j>i+1)
  { m = (i+j)/2 ;
    if (A[m]-A[i] < m-i) {j=m} else {i=m}
  }
// INV plus j=i+1 geeft A[i+1] - A[i] < 1, dus A[i+1]=A[i]
return A[i]
```

Je krijgt een iets simpeler programma door m alleen met A[m] te vergelijken. Je zoekt dan eigenlijk een j met A[j]<j naast een i met A[i]>=i, want deze combinatie inpliceert dat A[i]=A[j].

```
i=1; j=n
// INV: i <= A[i] en A[j] < j
while (j>i+1)
  { m = (i+j)/2 ;
    if (A[m] < m) {j=m} else {i=m}
  }
// INV plus j=i+1 geeft i <= A[i] en A[i+1] < i+1
// dus A[i+1]=A[i]
return A[i]
```

In beide oplossingen zie je het algemene stramien van binair zoeken: kies een m midden tussen i en j, en vervang een van deze door m.

Bespreking van antwoorden.
* Enkele slimmeriken hadden bedacht dat je van een segment van lengte 3, de middelste meteen kunt opleveren; erg leuk.
* Maak het niet te ingewikkeld! "i + floor((j-i)/2)" is echt niet duidelijker dan floor((i+j)/2), maar wel hetzelfde. De waarde m= (j-i)/2 valt buiten het bereik i..j.
* De meesten hadden wel door dat je een segment kunt opdelen en dan, door naar het midden te kijken, ontdekken of je links of recht verder moet zoeken. Maar wat moet je daar in het midden voor doen? Vergelijken met index m is een aardig

idee. En wat moet je dan doen bij gelijkheid? Dit is een vraag die je het snelst beantwoordt met een ASSERTIONELE kijk op het probleem, want je invariant geeft meteen antwoord.
* Wat je niet moet doen is testen op gelijkheid: A[m]==m. Mogelijk is deze fout ingegeven door de gedachte dat alle getallen tenminste eenmaal voorkomen; dat blijkt echter totaal niet uit het gegeven. Kijk maar eens naar deze invoer voor n=7:

```
1 3 4 5 6 6 6
```

In het midden geldt ONgelijkheid, je gaat "dus" naar rechts en daar is geen dubbele meer te vinden.
Ook deze fout is terug te voeren op een OPERATIONELE kijk op wat een algoritme is. Je schrijft wat code, en test die met de hand op enkele kleine gevallen, waarbij je de gegevens teveel interpreteert. Het is dan zo dat bij invoeren, waar WEL elk getal voorkomt, tot aan de eerste dubbel, WEL de gelijkheid i=A[i] geldt, zoals hier:

```
1 2 3 4 5 5 6
```

Maar nogmaals, dit was totaal niet gegeven.
* Een recursieve oplossing is niet compleet zonder eerste (buitenste) aanroep. Vooral als er in de body dingen worden berekend die weer mee worden gegeven.
* Een recursieve aanroep moet ook voorzien zijn van een beschrijving van de betekenis van parameters en van wat er precies opgeleverd wordt (een index of een getal bv.)
Dat kan er bijvoorbeeld uitzien als

```
zoekDub(i,j)
  // i en j zijn indices waarvoor A[j]-A[i] < j-i
  // levert op index h zdd A[h] = A[h+1] en i <= h <= j
```

De rol van een parameter/return beschrijving is vergelijkbaar met een invariant in een iteratief algoritme, dwz., onmisbaar.
* Houd het simpel, houd het simpel, houd het simpel!
Vaak zag ik dat er rond het midden getest werd op een dubbele, als in A[m]==A[m+1] || A[m-1]==A[m], waarna dan alleen bij het falen verder werd gezocht in (i,m-1) of (m+1,j). In het algemene geval kun je je beter concentreren op het halveren, de oplossing vind je dan vanzelf op het eind. Ga niet kijken of er bij je halveringsstap misschien "toevallig" een basis-gevalletje verstopt zit.
* Als je operationeel denkt en door herhaald testen van gevallen het probleem te lijf gaat, ga je vanzelf doorstrepen: een < wordt <=, dan toch maar met m-1 ipv m vergelijken, etc. Werk je met een invariant die helder vertelt WAT je van i en j verwacht, dan is doorstrepen niet nodig omdat je vanzelf ziet of m de i- of de j-achtige eigenschap heeft.
* Oplossingen die lineaire tijd gebruiken krijgen 0pt.

Dat het in O(lg n) werkt was bij de meeste constructies wel duidelijk; je kunt hiervoor opmerken dat de waarde j-i in elke ronde halveert en dus als variant kan worden gebruikt om het termineren in lgn ronden aan te tonen. Voor toekenning van alle 4 de punten moet je zeggen (liefst in de vorm van een invariant) waarom je algoritme werkt.

Om te kijken of je het nu beter begrepen hebt: Ga uit van A[1]=1 en A[n]>n, en lever een waarde op die NIET voorkomt! Met een assertionele aanpak moet dit in 10 minuten kunnen.

Opgave 2. De lift.
(a) Formulering van de GCP.
De Greedy Choice Property is een essentieel onderdeel van het bewijs dat een zekere strategie tot een optimale oplossing leidt. Het doel ervan is dus niet om je algoritme nog eens te beschrijven, of om allerlei niet-interessante details van je algoritme of oplossing te herkauwen. Waar gaat het om? Je zoekt in een bepaalde zoekruimte naar EEN toegelaten oplossing van maximale waarde. Door het maken van een zekere keuze, sluit je een deel van je zoekruimte uit (in het voorbeeld: alle beladingen zonder het lichtste kind worden uitgesloten). Is het belangrijk dar daar geen optimale oplossing bij zit? Wat je in feite moet weten is: er is EEN optimale oplossing met de gemaakte keuze. Maar het is erg moeilijk om een zo algemeen geformuleerde eigenschap in 1x te bewijzen, omdat die optimale oplossing primo nog niet bekend is, en secundo het sowieso erg moeilijk is om van "een oplossing" optimaliteit te bewijzen.

Daarom krijgt de GCP een "algemene vorm" die stelt dat elke oplossing kan worden geevenaard (NIET PERSE overtroffen!) door een met de gemaakte keuze erin.
De algemene vorm van de GCP, "voor elke toegelaten oplossing is er een toegelaten oplossing die de greedy keuze maakt en minstens zo goed is", kun je hier vertalen naar
Voor elke toegelaten belading B
is er een toegelaten belading B'
die (1) a1 bevat, en (2) |B'| >= |B|.
Met name het tweede deel is cruciaal.
Eigenlijk moet je hier ook over de mogelijkheid van lege verzamelingen redeneren (de aanname dat er minstens 1 kind in de lift past, hoeft na een aantal reducties niet meer te gelden) maar als je hier stilzwijgend aanneemt dat B minstens 1 kind bevat is het goed. Het bewijs gaat dan zo:
Zij B een toegelaten belading, en a in B; neem B' = B\{a}+{a1}.
Nu (1) volgens constructie bevat B' a1;
(2) B' bevat evenveel elementen als B.
Omdat a1 <= a (eigenschap van a1) is gewicht B' <= gewicht B, dus omdat B toegelaten is, is B' dat ook. []

Het is niet nodig om voor a "het lichtste kind uit B" te nemen; je kunt in je bewijs een willekeurig kind vervangen door a1. De enige reden om ergens "de lichtse", "de eerste", oid te kiezen, is dat je later in je bewijs GEBRUIKT dat a lichter was dan al zijn vriendjes in B, maar dat is hier niet nodig.

Kijk nog eens naar het bewijs van Greedy-Activity-Selector, waar je WEL de "minimale" activiteit uit een oplossing gaat vervangen door a1: waar in het bewijs van stelling 16.1 wordt gebruikt dat ak de eerste activiteit was? Kent jouw bewijs voor "De Lift" een zelfde stap?

De FORMULERING van de GCP moet niet vermelden hoe je B' vindt uit B, alleen het noemen van de eigenschappen van B' hoort daar thuis.

(b) Reductie naar een kleiner probleem.
De beste toegelaten belading die 1 bevat, is gelijk aan {1}
plus de grootste belading uit {a2,...,an} die in M-a1 past.

(c) Minimaal aantal ritten?
Nee, probeer maar eens kinderen 50 50 60 70 te vervoeren
in een lift met capaciteit 120, dit kan in twee keer.

Een tegenvoorbeeld hier is eigenlijk de enige manier om aan te
tonen dat het ook echt niet werkt. Dat het "een beetje dom"
zou zijn om te veronderstellen dat "max aantal kinderen per
rit" vanzelf leidt tot "min aantal ritten" is duidelijk als
je een assertionele kijk op algoritmen aanneemt, omdat de ene
eigenschap niet direct de andere impliceert.

Als je hier een (overtuigend beargumenteerd) JA hebt gezegd,
is het erg instructief om jouw "bewijs" eens door te spitten
aan de hand van het tegenvoorbeeld hierboven, om te kijken
welke van jouw redeneerstappen tot de fout heeft geleid.

Opgave 3. Cykels in ongerichte graaf.
Een WOUD (Engels: Forest) is een graaf zonder cykels; dit kan
dus een BOOM zijn (samenhangende graaf zonder cykels) of een
onsamenhangende graaf waarvan elke samenhangscomponent een
boom is. Kortom, een woud bestaat uit een of meer bomen.
Nogal wat mensen dachten dat "woud" betekent: "onsamenhangend"
en de meesten daarvan probeerden dan, 2 dingen uit te vissen:
A of er cykels zijn, en B of er samenhang is. Maar je moest
dus maar 1 ding uitvissen namelijk A of er cykels zijn of non-A
er zijn geen cykels. En non-A is hetzelfde als: Woud.

Kijken naar het AANTAL kanten is niet voldoende, want een graaf
kan cykels bevatten ook als m < n. Neem bv. de graaf met
drie componentjes, nl. een driehoek plus twee maal een duo van
twee verbonden punten. Deze heeft 7 knopen, 5 kanten, 3
componenten en bevat een cykel (dus geen woud). Omgekeerd is
het overigens wel zo, dat de aanwezigheid van n of meer kanten
impliceert dat er een cykel is.

Cykels in een ongerichte graaf kun je vrij gemakkelijk vinden
met DFS: volgens de ongerichte klassificatie treedt er een
backedge op, die in de gerichte taal bestaat uit een TREE en een
BACK edge. Bij het exploreren van die kant ziet u de knoop v
GRIJS, ZONDER DAT v ZIJN VADER IS. Het kan ook vrijwel even
gemakkelijk met BFS.
Waar je op moet letten is, dat de TIJD van DFS/BFS kan oplopen
tot O(n+m), dus je mag niet (vanwege de gestelde tijdgrens) een
hele DFS of BFS ongeconditioneerd afmaken. Zodra je een cykel
vindt, moet je de zoektocht staken. Als er geen cykel is, is
m <= n-1 dus dan is de complete zoektocht wel acceptabel.

Let op de volgende zaken:
VADER: Als je bij DFS alleen let op grijsheid van een bekeken
knoop, zou je de backedge naar de vader als cykel aanmerken
wat niet de bedoeling is.

COMPONENT: Wanneer je, bij DFS of BFS, na het bouwen van 1 boom nog niet de hele graaf hebt gezien, moet je de andere componenten ook nog op cykels bekijken.

Opgave 4. Pad-Compressie in Union-Find.

(a) de UF kent Union, Make en Find.

(b) Padcompressie wordt gebruikt in de Find om daaropvolgende Finds sneller te maken.

(c) Find(x)

```
{ if (p[x] <> x) { p[x] = Find(p[x]) };
  return p[x]
}
```

Opgave 5. Depth First Search.

De start- en finishtijden van knopen vormen disjucte of geneste intervallen, maar nooit deels overlappende. Daarom kan

$du < dv < fu < fv$

nooit voorkomen, niet bij buren en niet bij nonburen.

Een kandidaat verwees hierbij zelfs naar de mooie Parenthesis Theorem die in het boek staat vermeld.

Veel kandidaten kwamen eerst met een mooie, maar lange, redenering voor buren u en v, soms nog uitgesplitst naar het geval dat v vanuit u werd ontdekt of niet. Om daarna een redenering te geven voor het algemene geval. Bedenk hierbij, dat een NEE voor het algemene geval direct impliceert dat het voor buren ook niet kan (speciaal geval), zodat je met 1 redenering kunt volstaan (wel zeggen dat die ook op buren van toepassing is).