Uitgebreide uitwerking tentamen Algoritmiek
Dinsdag 1 juni 2010, 10.00 – 13.00 uur

Opgave 1.

a. Een toestand is een rij van n posities (huizen), met op elke positie ofwel geen krant (bijv. aangegeven met een -) ofwel wel een krant (aangegeven met `k`). Ook is van belang wie er aan de beurt is. In de begintoestand zijn er nog nul kranten bezorgd; in een eindsituatie is in elk huis een krant afgeleverd.
Een actie is het zetten van een `k` op een lege positie (bezorgen van een krant in een lege brievenbus) of twee `k`'s, maar dan op twee naast elkaar gelegen posities (bezorgen van twee kranten, in twee naast elkaar gelegen huizen) door degene die aan de beurt is.

b. $n = 2$: meteen twee kranten bezorgen: `- -` $\longrightarrow$ `k k`
$n = 3$: bezorg 1 krant in het middelste huis: `- - -` $\longrightarrow$ `- k -`. De tegenstander kan nu maar 1 krant bezorgen, en de beginnende speler wint dus vervolgens.
$n = 4$: bezorg 2 kranten in de twee middelste huizen: `- - - -` $\longrightarrow$ `- k k -`. De tegenstander kan nu maar bij 1 huis een krant bezorgen, en de beginnende speler wint dus.

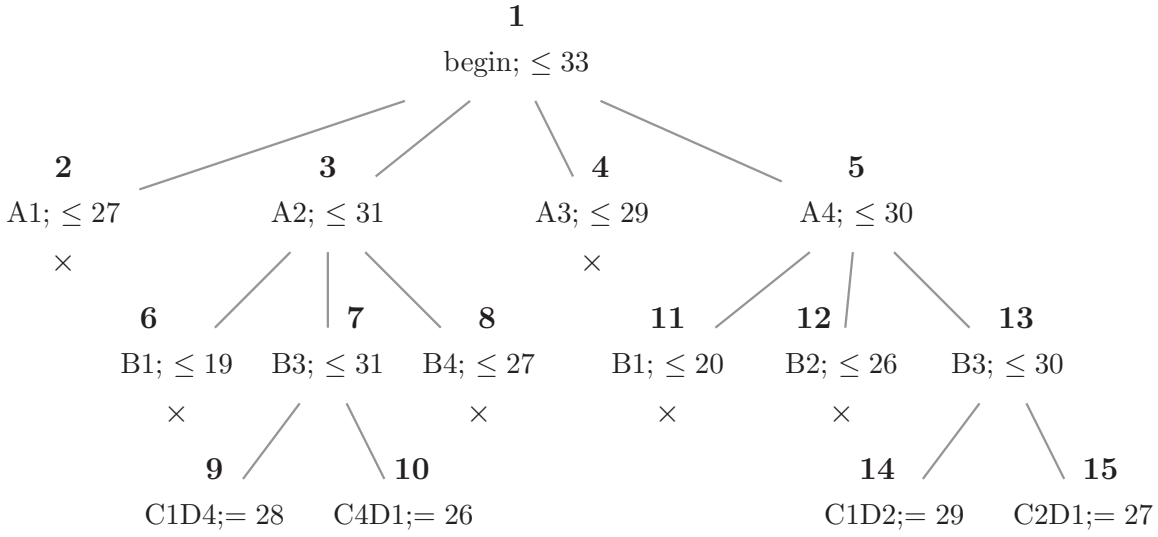
c. Zie het bijgevoegde bestand tad2010.pdf.
Uit de toestand-actie-ruimte, met per toestand aangegeven voor wie deze winnend is, zien we dat het spel met $n = 5$ winnend is (bij perfect spel) voor Bonny. De enige winnende zet is precies in het midden 1 krant bezorgen. Als Clyder er vervolgens twee naast elkaar bezorgt, wint Bonny meteen door bij de andere twee (naast elkaar gelegen) huizen een krant te bezorgen. Als Clyde er 1 bezorgt, zet Bonny op de een van de twee naast elkaar gelegen posities. Zij zal dan in haar volgende beurt winnen.

d. Winnende beginzet: als n oneven is bezorg je 1 krant in het midden, als n even is bezorg je twee kranten in de middelste twee huizen. Vervolgens heb je twee disjuncte rijtjes lege posities (huizen) die even groot zijn. De winnende strategie voor de speler die begonnen is (dat ben jij dus) is nu (na de bovenbeschreven beginzet): aap de zetten van de tegenstander na, maar dan in het andere rijtje. Dus als de tegenstander twee kranten bezorgt in het linkerrijtje, dan bezorg jij er twee op de corresponderende plekken in het rechterrijtje. Ten slotte zal hij een helft volmaken, waarop jij vervolgens de andere helft volmaakt en wint.
Voorbeeld: `- - - - - - - - - -` $\longrightarrow$ `- - - - k - - - -` $\longrightarrow$ `k k - - k - - - -` $\longrightarrow$ `k k - - k k k - -` $\longrightarrow$ `k k - - k k k - k` $\longrightarrow$ `k k - k k k k - k` $\longrightarrow$ `k k k k k k k - k` $\longrightarrow$ `k k k k k k k k k` .

Opgave 2.

a. Voorbeeld van een gretige strategie: kies combinaties (fabrikant, product) in volgorde van de kwaliteit, de combinatie met de grootste kwaliteit eerst. Als een combinatie in strijd blijkt met de eisen (dus fabrikant of product was al eerder gekozen), neem dan de volgende. Kies dus steeds de grootste die nog kan; hopelijk leidt dit tot een maximale toewijzing. Echter, dit algoritme levert voor het voorbeeld de toewijzing D1,B3,A2,C4 op, of D1,B3,C2,A4 (afhankelijk van hoe je methode omgaat met gelijke waardes; welke kies je?). Deze heeft totale kwaliteit 26 (resp. 27), en dat is niet optimaal. Het voorgestelde gretige algoritme werkt dus niet.
Een andere gretige strategie is bijv.: koppel de fabrikanten 1 voor 1 aan hun gunstigste product (dat met de hoogste kwaliteit), te beginnen met A, dan B, etc. Als een combinatie in strijd blijkt met de eisen (dus product was al eerder gekozen), neem dan het volgende product bij die fabrikant in grootte. Dit leidt tot A3,B2,C1,D4 (of A3,B4,C1,D2), met

totale kwaliteit 26. Ook niet optimaal.
b. Een best-fit-first *branch and bound*-algoritme gebruikt voor deeloplossingen (knopen in de state-space-tree) een afschatting (hier een bovengrens!) op de verwachte totale kwaliteit om enerzijds het zoeken naar een maximale oplossing te leiden (best-fit-first), en anderzijds te kunnen beslissen dat deeloplossingen niet verder uitgebreid hoeven te worden omdat het toch niet tot iets beters leidt. Als de huidige maximale waarde q bedraagt en de bovengrens in een knoop is $\leq q$, dan hoeft de deeloplossing/knoop niet verder bekeken te worden.
Oplossingen (hier toewijzingen) worden stapsgewijs opgebouwd. Bij uitbreiding van een knoop (deeloplossing) worden alle een-staps uitbreidingen daarvan gegeven, en voor elk daarvan wordt de bovengrens bepaald. In elke stap kiezen we de knoop (deeloplossing) met de hoogste bovengrens (best-fit-first) van alle nog niet gesnoeide knopen.
We genereren hier (deel)oplossingen door rij voor rij de fabrikanten af te lopen en deze te koppelen aan de producten (in de volgorde 1 t/m 4 bijv.) voor zover die nog niet in de koppeling (deeloplossing) voorkwamen. We nemen bijvoorbeeld als bovengrens voor de te verwachten totale kwaliteit: de totale kwaliteit van de betreffende deeloplossing + voor de andere fabrikanten (= andere nog te bekijken rijen) de waarde van het product met de grootste kwaliteitswaarde die we nog niet gehad hebben in de koppeling (deeloplossing). In de beginsituatie is een bovengrens voor de te verwachten kwaliteit: $7+9+8+9 = 33$. Deze waarde correspondeert niet met een goede oplossing, maar geeft wel een bovengrens. Voor de deeloplossing A2B4 is die bovengrens $5+5+8+9 = 27$. Voor A4 is de bovengrens: $4+9+8+9 = 30$. Zie verder de state-space-tree hieronder.



Opmerking. De niet-toelaatbare deeloplossingen zoals A2B2 zijn voor de duidelijkheid niet in de boom opgenomen. Dat soort knopen wordt toch niet verder uitgebreid. De dikgedrukte getallen bij de knopen geven de volgorde aan waarin de knopen worden *gemaakt en beoordeeld* (bovengrens berekend). De ×'s geven aan dat de knoop niet verder hoeft te worden uitgebreid.

Toelichting bij de state-space-tree: oplossingen worden stapsgewijs gegenereerd door een voor een de fabrikanten te koppelen aan productenn, te beginnen bij fabrikant A. Eerst wordt de beginknoop uitgebreid op alle mogelijke manieren (4 stuks): A1, A2, A3, A4. Van de corresponderende knopen wordt de bovengrens bepaald, en vervolgens wordt ver-

dergegaan met de meest veelbelovende knoop, zijnde A2 in dit geval. Deze wordt op alle mogelijke manieren uitgebreid (levert 3 toelaatbare deeloplossingen), waarvoor vervolgens de bovengrenzen worden berekend. Ga door met de knoop met de grootste bovengrens, in dit geval knoop B3. Deze kan op 2 manieren (toelaatbaar) worden uitgebreid; dit levert dan meteen twee oplossingen op, waarvan de beste totale kwaliteit 28 heeft. Ten gevolge daarvan kan nu op 3 plekken gesnoeid worden (die met bovengrens $\leq 27, 19, 27$), en er wordt verdergegaan met knoop A4. De uitbreidingen B1 en B2 kunnen meteen gesnoeid worden (want bovengrens ≤ 28). Alleen B3 wordt uitgebreid, en dit leidt tot de oplossing A4B3C1D2 met totale kwaliteit 29; de beste oplossing tot nu toe wordt ge-update en we hebben meteen de maximale oplossing gevonden, aangezien de enige nog hangende kandidaatknoop nu ook gesnoeid kan worden omdat die bovengrens ≤ 29 heeft.

Opgave 3.
a. Het initialiseren van de opa-velden kan bijv. met een postordewandeling:

```
void initialiseer(knoop* wortel) {
  if (wortel != NULL) {
    initialiseer(wortel->links);
    initialiseer(wortel->rechts);
    wortel->opa = NULL;
  }
} // initialiseer
```

b. Recursieve formulering: als je in een knoop zit laat je de opa-pointers van je kleinkinderen (indien die bestaan) naar jou toewijzen. Bijvoorbeeld: wortel->links->links->opa = `wortel`; wortel->links->rechts`opa` = wortel; wortel->links en wortel->links->links en wortel->links->rechts moeten dan wel alle drie $\neq$ `NULL` zijn. Andere gevallen analoog.

```
void grootvader(knoop* wortel) {
  if (wortel != NULL) {
    // boom niet leeg (*)
    if (wortel->links != NULL) {
      if (wortel->links->links != NULL)
        wortel->link->links->opa = wortel;
      if (wortel->link->rechts != NULL)
        wortel->link->rechts->opa = wortel;
    }
    if (wortel->rechts != NULL) {
      if (wortel->rechts->links != NULL)
        wortel->rechts->links->opa = wortel;
      if (wortel->rechts->rechts != NULL)
        wortel->rechts->rechts->opa = wortel;
    }
    grootvader(wortel->links);
    grootvader(wortel->rechts);
    // deze aanroepen mogen ook op plek (*)
  } // boom niet leeg
} //  grootvader
```

c. Volg de opa-pointers tot NULL. Dan zit je ofwel in de wortel, ofwel in een kind van de wortel. Als de opa-pointer van je kinderen NULL zijn zit je in de wortel, anders in een kind van de wortel.

```
int root(knoop* wijzer) {
  knoop* loper = wijzer;
  while (loper->opa != NULL)
    loper = loper->opa;
  // nu wijst loper naar de wortel of een kind daarvan
  if ((loper->links != NULL)&&(loper->links->opa != NULL))
    return loper->links->opa->info;
  if ((loper->rechts != NULL)&&(loper->rechts->opa != NULL))
    return loper->rechts->opa->info;
  // anders zat je blijkbaar al in de wortel
  return loper->info;
} // nivo
```

Opgave 4.

a. Het array bevat n plaatsen. Er moeten hooguit $m < n$ waardes in het array staan. Als alle (maximaal) m waardes maar 1 keer voorkomen, worden slechts (maximaal) $m < n$ plekken bezet. Dus minstens 1 waarde komt ≥ 2 keer voor.

b. We brengen het geval $A[0], \ldots, A[n-1]$ terug tot $A[0], \ldots, A[n-2]$. Of algemener: we brengen het geval $A[0], \ldots, A[i-1]$ terug $A[0], \ldots, A[i-2]$. Als we zeker weten dat er een dubbele zit in $A[0], \ldots, A[i-1]$, dan is dat ofwel de laatste (en dan $A[i-1] == A[i-2]$, want A is oplopend gesorteerd), ofwel een dubbel element zit in het voorstuk $A[0], \ldots, A[i-2]$.

```
int dubbel1(int A[], int i) {
  // bij aanvang weten we zeker dat er een dubbele in zit
  if (i==2)
    return A[i-1];
  // het dubbele element moet dus A[1] (==A[0]) zijn
  else {
    if (A[i-1] == A[i-2])
      return A[i-1];
    // A[i-1] komt dubbel voor
    else
    // dan moet de dubbele in het voorstuk zitten
    // merk op dat we daarom bij elke recursieve aanroep zeker weten dat er een
    // dubbele in het te bekijekn deelarray zit
      return dubbel dubbel1(A,i-1);
  }
} // dubbel1
```

c. We splitsen het array nu (herhaald) in twee stukken. Dan zit een dubbel element ofwel in het midden, ofwel in de linkerhelft (recursieve aanroep) ofwel in de rechterhelft (recursieve aanroep). We weten nu bij een recursieve aanroep niet altijd zeker dat er een dubbele waarde in voorkomt. Daarmee moeten we dus rekening houden.
Merk op dat het basisgeval wordt gegeven door $r = l + 1$ als we het stuk $A[l], \ldots, A[r]$ bekijken.

```
int dubbel2(int A[], int l, int r) {
  if (r == l+1) { // 2 elementen
    if (A[l]==A[r])
      return A[l];
    else
      return -1;
      // -1 betekent dus: geen dubbele aanwezig
  } // basisgeval twee elementen
  else {
  // meer dan twee elementen: recursieve aanroepen
    int m = (l+r)/2;
    if (A[m] == A[m+1])
      return A[m];
    else {
      int hulp = dubbel2(A,l,m);
      if (hulp != -1)
        // er zit een dubbele links
        return hulp;
      else
      // zo niet, dan rechts kijken
        return dubbel2(A,m+1,r);
    } / else recursieve aanroepen
  } // else meer dan 2 elementen
} // dubbel2
```

d. Eerst kijken we rond het midden, dus of $A[m] = A[m+1]$. Als daar geen dubbele zit, zit ofwel links ofwel rechts (of in beide) een dubbele. Dit is onder andere te zien aan het aantal elementen en de waardes die in een deelarray zitten, dus analoog als in **a.**. Het deelarray $A[i], \ldots, A[j]$ $(j > i)$ bevat $j - i + 1$ plekken. Alle waardes in dat stuk zitten tussen $A[i]$ en $A[j]$ in. Dus dat zijn maximaal $A[j] - A[i] + 1$ verschillende waardes. Als $A[rj] - A[i] + 1 < j - i + 1$ weet je zeker dat er in dat deelarray een dubbele waarde zit. Anders $(A[rj] - A[i] + 1 \geq j - i + 1$ hoeft dat niet. In dat geval zit er zeker rechts een dubbele, omdat we daar dan minder verschillende waardes hebben dan er plaatsen zijn. We roepen dubbel3 alleen aan op een deelarray als we zeker weten dat het deelarray een dubebele waarde bevat.

```
int dubbel3(int A[], int l, int r) {
// we weten zeker dat het deelarray een dubbele waarde bevat
  if (r == l+1) {
  // voor twee elementen is dan dus A[l]==A[r]: A[l] komt dubbel voor
      return A[l];
  } // basisgeval twee elementen
  else {
  // meer dan twee elementen: slechts 1 recursieve aanroep is nodig
    int m = (l+r)/2;
    if (A[m] == A[m+1])
      return A[m];
    else {
    // een/het dubbel element zit niet in het midden, dus links of rechts
```

```
    // we kunnen zeker weten in welke helft in elk geval een dubbele zit
      if (A[m]-A[l]+1 < m-l+1)
        // er zit zeker een dubbele links
        return dubbel3(A,l,m);
      else
      // zo niet, dan moet er rechts een dubbele zitten
        return dubbel3(A,m+1,r);
    } // else recursieve aanroepen
  } // meer dan 2 elementen
} // dubbel3
```

Opgave 5. Werking van het algoritme van Dijksta op het voorbeeld:
1. Geef knoop 1 label 0.
2. Selecteeer knoop 1. Pas de labels van de knopen aangrenzend aan 1 aan, die nog niet in U gekozen zijn. Label knoop 2 wordt 5; label knoop 6 wordt 3.
2. Selecteer de knoop met het kleinste label, dus selecteer 6 (en tak (1,6)). Zijn label is nu de definitieve kortste afstand vanaf 1. Pas de labels aan: knoop 4 krijgt label 4 en knoop 7 label 7.
3. Selecteer knoop 4 (en tak (6,4)). Pas de labels aan: label knoop 2 blijft 5, knoop 5 krijgt nu label 10
4. Selecteer 2 (en tak (1,2)). Labels aanpassen: label 5 wordt nu 8, 3 krijgt label 13.
5. Selecteer 7 (en tak (6,7)). Labels aanpassen: label 5 blijft 8, 8 krijgt label 10.
6. Selecteer 5 (en tak (2,5)). Labels aanpassen: label 3 blijft 13, 9 krijgt label 11, label 8 wordt 9.
7. Selecteer 8 (en tak (5,8)). Labels aanpassen: label 3 wordt 12, label 9 wordt 10.
8. Selecteer 9 (en tak (8,9)). Labels aanpassen: label 3 wordt 11.
9. Selecteer 3: klaar!

De boom van kortste paden: de nummers van de knopen in de boom corresponderen met de knoopnummers in de graaf; de dikgedrukte getallen naast de knopen stellen de kortste afstand vanaf de beginknoop voor.