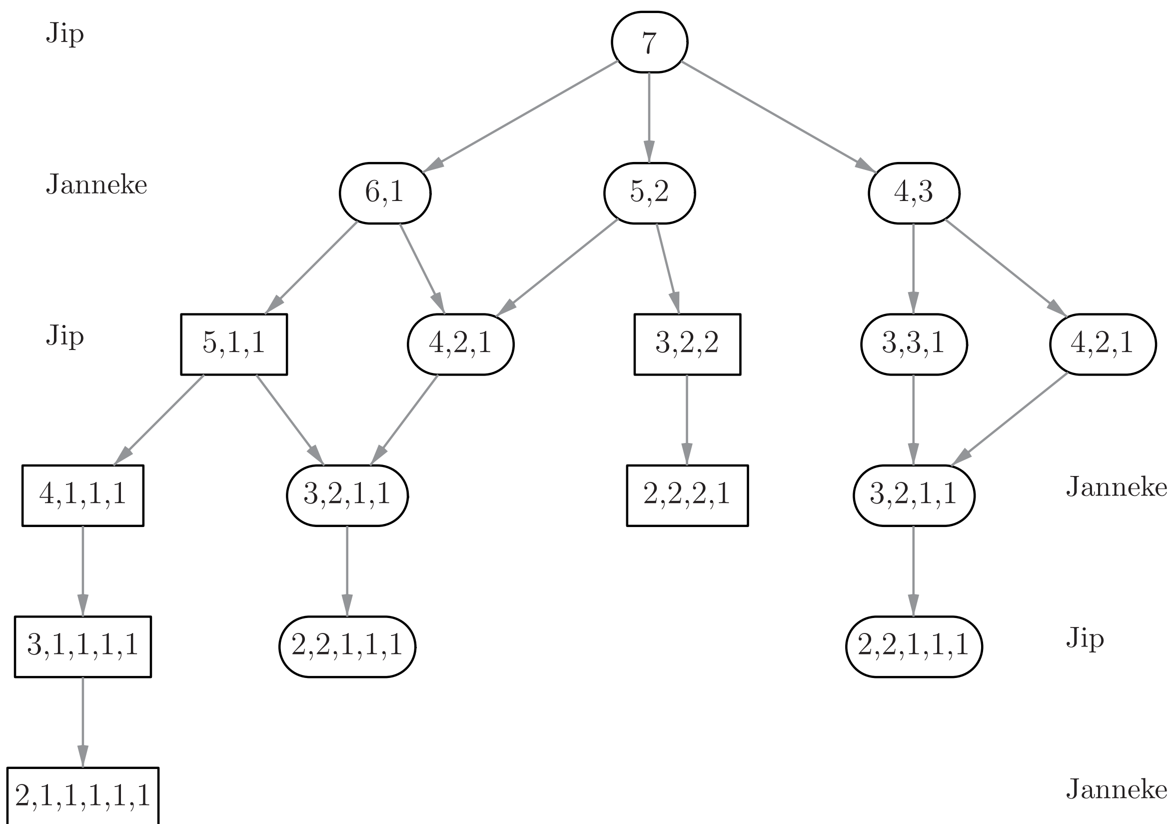
Uitgebreide uitwerking tentamen Algoritmiek
Dinsdag 5 juni 2007, 10.00 – 13.00 uur

Opgave 1.
a. Een toestand bestaat hier uit een aantal stapels, met op elk van die stapels een aantal munten (hooguit n per stapel). In de begintoestand is er één stapel van n munten, in een eindsituatie zijn er alleen nog stapels met 1 munt of met 2 munten.
(Eigenlijk maakt ook de speler die aan de beurt is deel uit van een toestand: immers bijvoorbeeld 3, 2, 1, 1 is winnend voor Janneke als zij aan de beurt is, maar verliezend voor haar als Jip aan de beurt is. Het maakt dus uit wie er aan de beurt is.)
Een actie is het doen van een zet door degene die aan de beurt is: dus het splitsen van een van de bestaande stapels in twee stapels van ongelijke grootte.
b. Merk op dat een stapel van 2 niet meer gedeeld kan worden volgens de spelregels, stapels van meer munten wel. Dus een eindstand bevat alleen stapels van 1 en 2. Het grootste aantal zetten krijg je als er zoveel mogelijk stapels van 1 worden gemaakt. Dat gebeurt als je eerst de stapel van n verdeelt in één van 1 en één van $n-1$, vervolgens die van $n-1$ in één van 1 en één van $n-2$, en zo verder steeds de stapel van $m > 2$ verdelen in één van 1 en één van $m-1$, totdat je $n-2$ stapels van 1 overhebt en één van 2. Dat kost je in totaal $n-2$ zetten. Dit is dus het maximale aantal zetten dat kan voorkomen.
c+d.



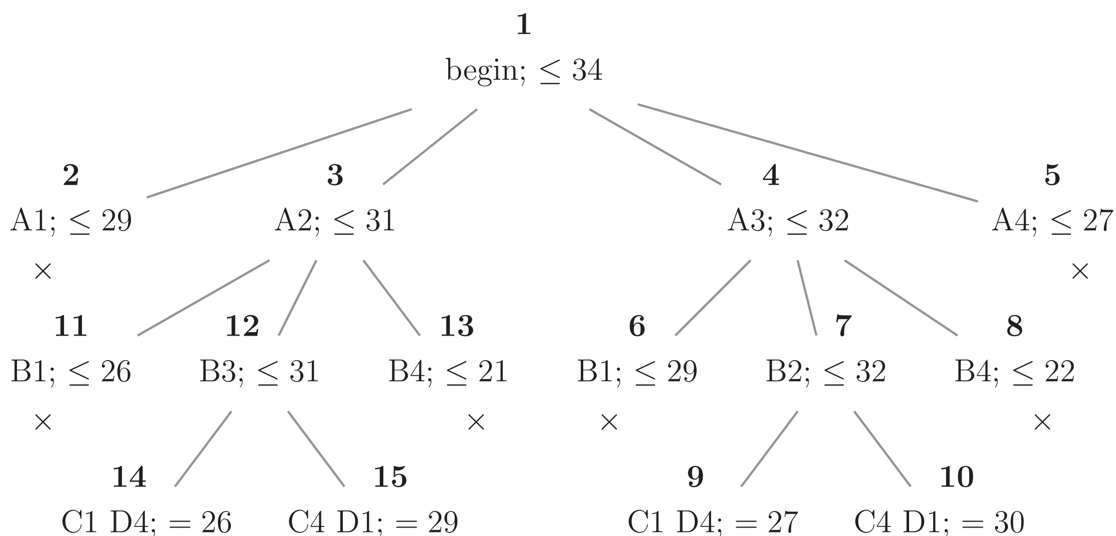
Links dan wel rechts naast de toestand-actie-ruimte staat aangegeven wie er op elk niveau aan de beurt is.
Toestanden aangegeven met een rechthoek zijn winnend voor Jip, de andere (ovaal) zijn winnend voor Janneke. Aangezien Jip begint en alle drie vervolgtoestanden winnend zijn

voor zijn tegenstander, kan hij het spel bij perfect spel van Janneke niet winnen. Het spel is dus winnend voor Janneke.

Opgave 2.

a. Antwoord 1: genereer alle mogelijke toewijzingen. Dit komt neer op het genereren van alle mogelijke permutaties van de producten 1 t/m n. (De permutatie 3 2 4 1 betekent dan dat fabrikant A product 3 maakt, B maakt 2, C maakt 4 en D maakt 1.) Dat zijn dus $n!$ mogelijke toewijzingen. Bereken van elke toewijzing de totale kwaliteit en houd steeds de tot dusver maximale bij.
Antwoord 2: genereer alle n^n mogelijke combinaties van producten en fabrikanten: voor elke fabrikant zijn er n mogelijke producten, dus n fabrikanten koppelen aan n producten geeft n^n mogelijkheden. Van elke mogelijke combinatie wordt gecontroleerd of het een goede toewijzing is (elk product komt precies één keer voor), en van de goede toewijzingen wordt de maximale bepaald. Verschil met Antwoord 1. is dat hier bij het genereren van de combinaties geen rekening wordt gehouden met de restrictie dat elk product maar door één fabrikant wordt geproduceerd. Dat wordt pas achteraf gecontroleerd.
b. Antwoord 1.a. (hoort bij Antwoord 1. uit **a.**): we genereren de permutaties stap voor stap door steeds aan de volgende fabrikant de producten een voor een toe te kennen, waarbij we nooit een product kiezen dat al aan een eerdere fabrikant gekoppeld is. Echter omdat we een maximum zoeken moeten alle $n!$ toewijzingen bekeken worden, en het heeft ook geen zin om de kwaliteit van de deeloplossing die we aan het uitbouwen zijn te vergelijken met de tot dusver maximale. Als we uitbreiden kan de waarde immers nog groter worden, dus we moeten doorgaan met uitbreiden. Backtracking is in dit geval alleen maar een manier waarop de permutaties worden gegenereerd (nl. stap voor stap). De geschetste methode is dus niet beter dan de exhaustive search methode.
Als een minimum gezocht werd heeft het overigens wel zin om de waarde van de deeloplossing te vergelijken met de tot dusver minimale: als je erboven zit hoef je niet verder meer uit te breiden.
Antwoord 1.b. (hoort bij Antwoord 1. uit **a.**): er is trouwens wel een eenvoudige test mogelijk die we kunnen gebruiken tijdens de stap-voor-stap constructie van toewijzingen als hierboven, en die ook snoeit op grond van het niet meer kunnen bereiken van een betere waarde. Aangezien de kwaliteitwaarden `quality`$[i][j]$ tussen 1 en 10 zitten kun je concluderen dat de uiteindelijk te behalen totale kwaliteit altijd kleiner of gelijk zal zijn aan de kwaliteit van je deeltoewijzing + (het aantal nog te koppelen fabrikanten)$*10$. Als deze waarde kleiner is dan de tot dusver gevonden totale kwaliteit, dan hoef je niet verder uit te breiden. Dit backtracking algoritme is i.h.a. wel beter dan het exhaustive search algoritme uit a.
Antwoord 2. (hoort bij Antwoord 2. uit **a.**): We genereren de toewijzingen stap voor stap door telkens de volgende fabrikant aan een product te koppelen, en controleren al tijdens de constructie of het een correcte toewijzing is door te checken of het gekozen product al geweest is of niet. Zo ja dan koppelen we de volgende fabrikant aan een product, zo nee dan herzien we de keuze. Als alle producten in een stap geprobeerd zijn herzien we de keuze bij de vorige fabrikant. We zijn hier eigenlijk bezig stap voor stap met backtracking permutaties te genereren. Dit is efficiënter dan exhaustive search omdat we niet alle n^n mogelijkheden volledig hoeven te genereren, maar vroegtijdig foute constructies afbreken. In essentie worden nu slechts $n!$ mogelijkheden gegenereerd.
Tijdens de constructie kan ook de kwaliteit van de deeloplossing worden bijgehouden en ge-update als deze wordt uitgebreid. Echter aangezien een maximum wordt gezocht

kun je niet zomaar stoppen met de constructie op grond van vergelijken van de huidige maximale totaalkwaliteit met de kwaliteit van de deeloplossing. Vergelijk de overwegingen onder Antwoord 1.a. en 1.b.
c. Een best-fit-first *branch and bound*-algoritme gebruikt 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 is $\leq q$, dan hoeft de deeloplossing/knoop niet verder bekeken te worden. In dit geval nemen we bijvoorbeeld als bovengrens voor de te verwachten totale kwaliteit: de kwaliteit van de betreffende deeloplossing + uit elke nog te bekijken rij/fabrikant de grootste waarde (uit kolommen/producten die we nog niet gehad hebben). Dus in de beginsituatie is een bovengrens voor de te verwachten kwaliteit: $8+9+8+9$. Voor de deeloplossing A2 B3 is die bovengrens $14+8+9=31$. In elke stap van het algoritme bekijken we de meest veelbelovende (= met de hoogste bovengrens in dit geval) deeloplossing, breiden die op alle mogelijke manieren uit (zie boom hierna), berekenen de bovengrens van die uitbreidingen en kiezen dan uit *alle* deeloplossingen weer de meest veelbelovende, etcetera. Het uitbreiden verloopt dus ook heel anders dan bij backtracking, waarbij steeds één deeloplossing steeds verder wordt uitgebreid tot deze is afgehandeld.



Opmerking. De niet-toelaatbare deeloplossingen zoals A2 B2 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 $\times$'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 producten, 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 (bijv. voor A1: $3+9+8+9=29$), en vervolgens wordt verdergegaan met de meest veelbelovende knoop, zijnde A3 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 B2. Deze kan op 2 manieren (toelaatbaar) worden uitgebreid; dit levert dan meteen twee oplossingen op, waarvan de beste kwaliteit 30 heeft. Ten gevolge daarvan kan nu op 4 plekken gesnoeid worden (die met bovengrens ≤ 30), en er wordt verdergegaan met knoop A2. Etcetera.

Opgave 3.

a.

```
void init(knoop* wortel) {
   if (wortel != NULL) {
      wortel->som = 0;
      init(wortel->links);
      init(wortel->rechts);
   }
} // init
```

Hier is een preorde-wandeling gebruikt, maar postorde (of symmetrische orde) mag natuurlijk ook.

b. Recursieve formulering: som(binaire boom):= som(linkersubboom) + som(rechtersubboom) + wortel->info, ofwel: wortel->som = wortel->links->som + wortel->rechts->som + wortel->info mits de som-velden in linker- en rechtersubboom alle reeds gevuld zijn. Dit leidt tot de volgende recursieve functie:

```
void optellen(knoop* wortel) {
   if (wortel != NULL) {
      optellen(wortel->links);
      optellen(wortel->rechts);
      if (wortel->links != NULL)
         wortel->som += wortel->links->som;
      if (wortel->rechts != NULL)
         wortel->som += wortel->rechts->som;
      wortel->som += wortel->info;
   }
} // optellen
```

Opgave 4.

a. Divide and conquer: verdeel het array in een linkerhelft en een rechterhelft; zoek links naar een index i waarvoor $A[i] = i$ en idem rechts (*recursie*); als zo'n index gevonden is deze retourneren, anders 0 teruggeven.

Aangezien door de recursie steeds een ander, kleiner stuk van het array wordt doorzocht schrijven we (in pseudocode) een functie `zoeken(A,`l`,`r`)`, waarin `A[`l`]` t/m `A[`r`]` wordt bekeken. De functie wordt aangeroepen als: `zoeken(A, 1, `n`)`, met $n \geq 1$;

```
zoeken(A, l, r) ::
   if (l = r) then
      if (A[l]=l) then
         return l;
      else
         return 0; // niet gevonden
      fi
   else { // nu is l < r
```

```
        index1 := zoeken(A,l,⌊(l+r)/2⌋);
        // je kunt ook eerst hier testen of je de gevraagde
        // index gevonden hebt en dan alleen rechts
        // verder zoeken als deze links niet gevonden is.
        index2 := zoeken(A,⌊(l+r)/2⌋ + 1,r);
        if (index1 = 0) // links niet gevonden
           return index2;
        else
           return index1;
        fi
     fi .
```

Opmerking: je kunt natuurlijk ook eerst in het `midden` $= \lfloor \frac{l+r}{2} \rfloor$ kijken, en dan ben je klaar als `A[midden]=midden`, en anders ga je links van het midden en rechts van het midden verder zoeken. Zie ook onderdeel **b.** De essentie is dat het probleem ter grootte n wordt opgesplitst in twee keer hetzelfde probleem ter grootte ongeveer $\frac{n}{2}$.
b. Decrease by half: het probleem ter grootte n wordt gereduceerd tot (één versie van) hetzelfde probleem ter grootte $\frac{n}{2}$. Als bij binair zoeken kunnen we ons beperken tot het zoeken in ofwel de linkerhelft, ofwel de rechterhelft van het array, op grond van het feit of `A[midden]` > 0 of < 0 is. Dit moet wel bewezen worden. In het bewijs hieronder wordt gebruikt dat `A` oplopend gesorteerd is, dat alle array-elementen verschillend zijn en dat ze geheeltallig zijn.

Laat `midden` $= \lfloor \frac{l+r}{2} \rfloor$. Als `A[midden]` $>$ `midden`, dan is `A[`j`]` $> j$ voor alle $j >$ `midden`. Immers, `A[`j`]` neemt als j met 1 toeneemt ook met minstens 1 toe: `A[`$j+1$`]` $\geq$ `A[`j`]` $+1$, en dus is ook `A[`$j+k$`]` $\geq$ `A[`j`]` $+k$ ($k > 0$). En derhalve is `A[`$m+k$`]` $\geq$ `A[`m`]` $+k > m+k$ als `A[`m`]` > 0. Q.E.D.

Derhalve hoeven we in dit geval niet in de rechterhelft verder te zoeken, alleen in de linkerhelft. Analoog: Als `A[midden]` $<$ `midden`, dan is `A[`j`]` $< j$ voor alle $j <$ `midden` en dus hoeven we in dat geval alleen rechts verder te zoeken.

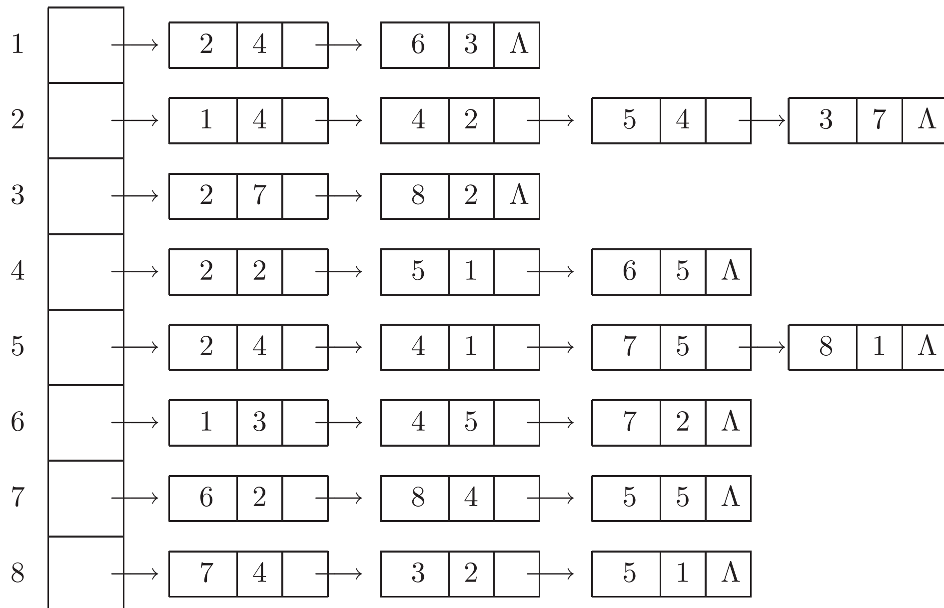
```
zoek(A, l, r) ::
   if (l ≤ r) then
      midden := ⌊(l+r)/2⌋ ;
      if (A[midden]=midden) then
         return midden;
      else
         if (A[midden] > midden) then
            return zoek(A,l,midden-1);
         else
            return zoek(A,midden+1,r);
         fi
      fi
   fi .
```

Opgave 5.

a. Adjacency list representatie in C++:

```
struct buur {
  int knoopnummer;
  int gewicht;
  buur* volgende;
}
buur* inhoud[n];
```

Adjacency list voor de voorbeeldgraaf:



b. Werking van het algoritme van Dijksta op het voorbeeld:

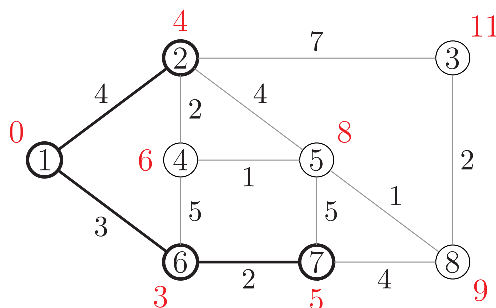
1. Geef knoop 1 label 0. Selecteeer (= stop in U) knoop 1. Pas de labels van de knopen aangrenzend aan 1 aan, die nog niet in U zitten. Label knoop 2 is nu 4; label knoop 6 is 3.

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 8 en knoop 7 label 5.

3. Selecteer knoop 2 (en tak (1,2)). Pas de labels aan: knoop 3 krijgt label 11, knoop 4 krijgt nu label 6 (label aangepast: via knoop 2 is de afstand korter) en knoop 5 krijgt label 8.

4. Selecteer 7 (en tak (6,7)). Labels aanpassen: 5 houdt label 8 en 8 krijgt label 9.

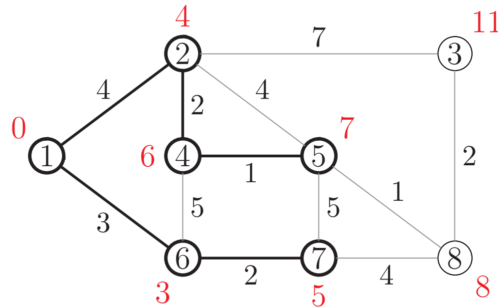
Tussenresultaat:



Dikgedrukt de reeds geselecteerde knopen en de takken die corresponderen met de tot dusver gevonden kortste paden. Bij de knopen staat (in rood) het label van de knoop, voorzover deze niet ∞ zijn. Voor de geselecteerde knopen is het label precies de kortste afstand vanaf knoop 1.

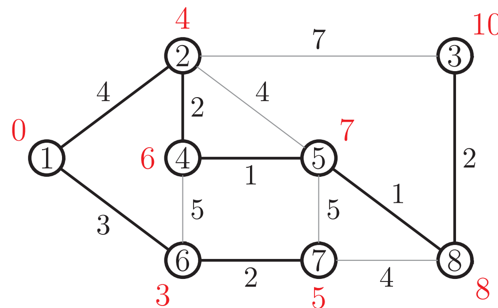
5. Selecteer vervolgens knoop 4 (en tak (2,4)) en pas de labels aan. Het label van 5 wordt veranderd in 7.
6. Selecteer 5 (en tak (4,5)) en pas de labels aan. Knoop 8 krijgt nu label 8 (via 5 dus).

Tussenresultaat:



7. Selecteer nu knoop 8 (en tak (5,8)) en pas labels aan. Het label van 3 wordt nu veranderd in 10
8. Kies tenslotte knoop 3. In onderstaand plaatje geven de labels van alle knopen nu de kortste afstanden aan vanuit knoop 1.

Eindresultaat:



De dikgedrukte takken vormen samen de boom van kortste paden.