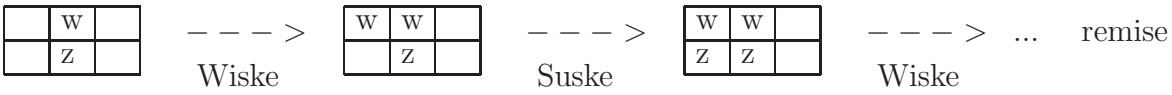
Uitgebreide uitwerking tentamen Algoritmiek
Dinsdag 3 juni 2008, 10.00 – 13.00 uur

Opgave 1.
a. Een toestand is hier een m bij n bord met voor elk vakje aangegeven of het leeg is, óf een witte steen bevat óf een zwarte steen. Verder is van belang wie er aan de beurt is, dus dat wordt ook opgenomen in de toestand. In de begintoestand is het bord leeg, in een eindsituatie is het bord helemaal vol met stenen of kan een van beide spelers niet meer zetten.
Een actie is het doen van een zet door degene die aan de beurt is: dus het plaatsen van een steen op een leeg vakje, horizontaal of verticaal grenzend aan een vakje met een eigen steen.
b.+ c. Zie het bijgevoegde bestand statespace.pdf.
Rechts naast de toestand-actie-ruimte staat aangegeven wie er op elk niveau aan de beurt is. Wiske speelt met wit (w) en Suske met zwart (z).
Toestanden aangegeven met een rechthoek zijn winnend voor Wiske, de andere (ovaal) zijn remisestanden. Toestanden met winnaar Suske komen in dit plaatje niet voor. Aangezien Wiske begint en een van de drie vervolgtoestanden winnend is voor haar, zal zij bij perfect spel winnen. Het spel is dus winnend voor Wiske. Zij moet dus in haar eerste zet onder haar eigen steen zetten (dus midden onder). Suske heeft dan nog maar één mogelijke zet. Vervolgens maakt het niet uit welke van de twee mogelijke zetten Wiske doet, beide leiden tot winst: Suske is immers geblokkeerd en kan niet meer zetten.
d. Uit **b.+ c.** volgt: als Suske rechtsonder zet verliest hij. Uit symmetrie-overwegingen is de zet linksonder ook verliezend. Als Suske linksbover zet kan Wiske hem blokkeren door midden onder te zetten. Dus dan verliest Suske ook (bij perfect spel van Wiske). Uit symmetrie-overwegingen leidt ook de zet rechtsboven tot winst voor Wiske. Resteert voor Suske als eerste zet nog de zet midden onder. In dat geval mag Wiske alleen linksboven of rechtsboven zetten. Als Suske vervolgens steeds onder de laatst gezette steen van Wiske zet, wordt het remise. Zie plaatje hieronder. (Overigens maakt het zelfs niet uit welke zet Suske doet als tweede en derde zet; beide mogelijkheden leiden tot remise.)



Opgave 2.
a. Recursieve formulering:
wortel->hoogte = max (wortel->links->hoogte, wortel->rechts->hoogte) + 1; wortel->links en wortel->rechts moeten dan wel $\neq$ `NULL` zijn, en hun hoogtevelden moeten bekend zijn. Dit leidt tot een postordewandeling.
Hieronder is `max` een functie die het maximum van zijn argumenten berekent.

```
void vulhoogte(knoop* wortel) {
  if (wortel != NULL) {
    vulhoogte(wortel->links);
    vulhoogte(wortel->rechts);
    if ((wortel->links == NULL) && (wortel->rechts == NULL))
```

```
      wortel->hoogte = 1;
    else { if (wortel->links == NULL) // rechtersubboom nu dus niet leeg
             wortel->hoogte = wortel->rechts->hoogte + 1;
           else { if (wortel->rechts == NULL)  // linkersubboom niet leeg
                    wortel->hoogte = wortel->links->hoogte + 1;
                  else  // linker- en rechtersubboom beide niet leeg
                    wortel->hoogte = 1 +
                       max(wortel->links->hoogte, wortel->rechts->hoogte);
                }
         }
  }
} // vulhoogte
```

Opm. In plaats van geneste if-statements kun je natuurlijk ook telkens `return` gebruiken.

b. Recursieve formulering: gebalanceerd(binaire boom)= gebalanceerd(linkersubboom) AND gebalanceerd(rechtersubboom) AND abs(wortel->links->hoogte-wortel->links->hoogte) $\leq$ k. Dit leidt tot onderstaande recursieve functie. Hieronder is `abs` een functie die de absolute waarde van zijn argument berekent.

```
bool gebalanceerd(knoop* wortel, int k) {
  if (wortel == NULL)
    return true;
  // hier wortel niet NULL
  if (!(gebalanceerd(wortel->links,k)&&gebalanceerd(wortel->rechts,k)))
    return false;
  // hier ook links en rechts gebalanceerd
  if ((wortel->links == NULL) && (wortel->rechts == NULL))
    return true;
  if (wortel->links == NULL) // hier wortel->rechts niet NULL
    return (wortel->rechts->hoogte <= k);
  if (wortel->rechts == NULL) // hier wortel->links niet NULL
    return (wortel->links->hoogte <= k);
  // hier beide subbomen niet leeg!
  return (abs(wortel->links->hoogte - wortel->rechts->hoogte) <= k);
} // gebalanceerd
```

Opm. In plaats van `return`s te gebruiken kan het ook met geneste if-statements.

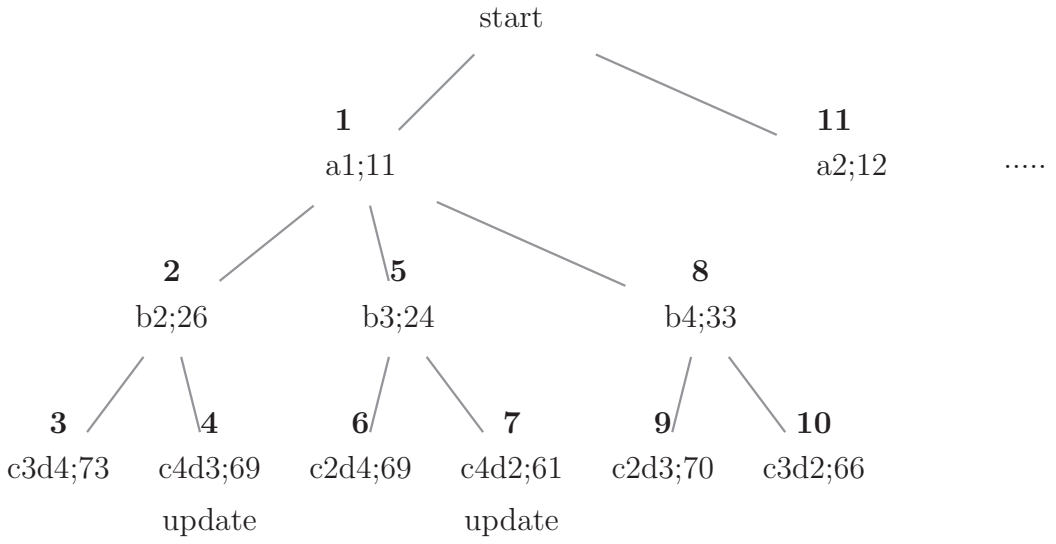
Alternatief voor **a.** (en je kunt geheel analoog een alternatieve oplossing voor **b.** geven):

```
void vulhoogte (knoop* wortel) {
  int hoogteL = 0;
  int hoogteR = 0;
  if (wortel != NULL) {
    if (wortel->links != NULL) {
      vulhoogte(wortel->links);
      hoogteL = wortel->links->hoogte;
    }
    if (wortel->rechts != NULL) {
```

```
      vulhoogte(wortel->rechs);
      hoogteR = wortel->rechts->hoogte;
    }
    wortel->hoogte = 1 + max(hoogteL,hoogteR);
  }
} // vulhoogte
```

Opgave 3.

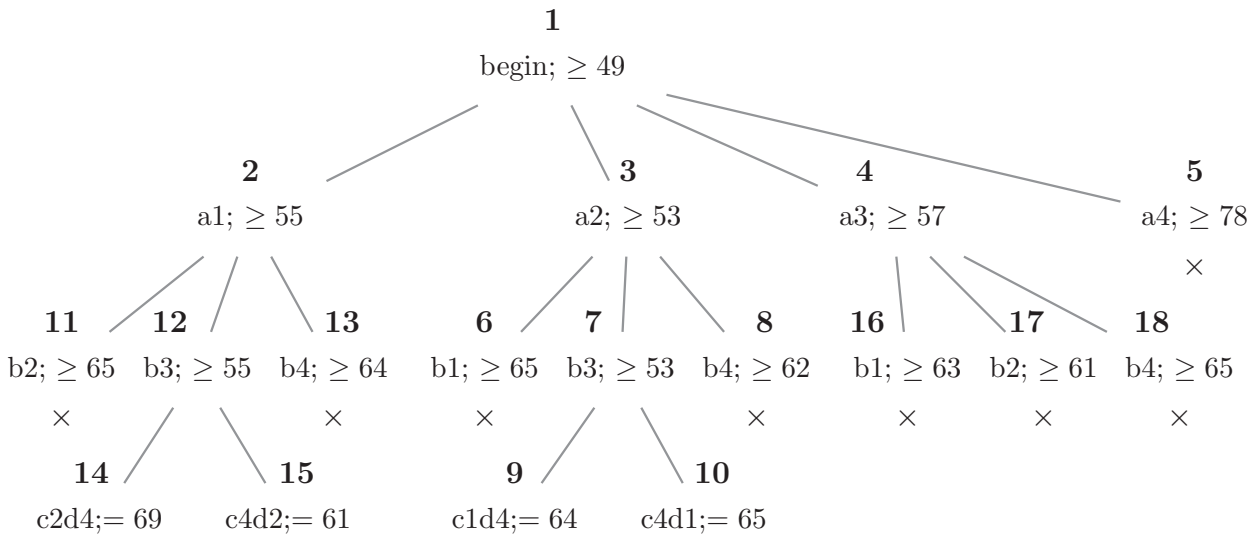
a. Genereer alle mogelijke bebouwingen stap voor stap door de gebouwen 1 voor 1 te koppelen aan een locatie. We beginnen met gebouw a, koppelen deze aan een locatie, dan gebouw b, dan gebouw c, etcetera. We proberen de locaties aan een gebouw te koppelen in de volgorde 1, 2, 3, enz. Een deeloplossing bestaat uit een deelkoppeling van de eerste gebouwen aan een unieke locatie, en wel zo dat elke locatie maar hooguit 1 keer voorkomt. Backtracking doet hier het volgende: een deeloplossing wordt uitgebreid door het volgende gebouw te koppelen aan de eerste locatie. Als deze locatie al voorkomt herzien we die keuze en proberen de volgende locatie (enz.). Als een locatie wel kan, gaan we de nieuwe deeloplossing op dezelfde manier uitbreiden. Als je alle locaties bij een gebouw geprobeerd hebt ga je terug naar het vorige gebouw en probeer je daar de volgende locatie. Verder houd je van je deeloplossing bij en je onthoudt de kosten van de tot dusverre gevonden goedkoopste volledige oplossing. In elke stap vergelijk je de deelkosten met de tot nu toe goedkoopste. Als de deelkosten hoger of gelijk zijn dan hoef je op die weg niet verder te gaan (dus deeloplossing niet verder uitbreiden) en herzie je je laatste keuze. Zodra je een volledige oplossing gevonden hebt update je de minimale kosten indien nodig.



In de state space zijn de deeloplossingen die meteen herzien zijn omdat de locatie al geweest is, zoals a1b1 en a1b2c2, weggelaten om het plaatje overzichtelijk te houden. Hier is gebacktrackt omdat eenzelfde locatie 2 keer voorkomt. Het backtrackingalgoritme moet in dit geval alle deeloplossingen bekijken.

b. Een best-fit-first *branch and bound*-algoritme gebruikt een afschatting (hier een ondergrens!) op de verwachte totale kosten om enerzijds het zoeken naar een minimale 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 minimale waarde q bedraagt en de ondergrens is $\geq q$, dan hoeft de deeloplossing/knoop niet verder bekeken te worden.
In dit geval nemen we bijvoorbeeld als ondergrens voor de te verwachten totale kosten: de kosten van de betreffende deeloplossing + uit elke nog te bekijken rij/gebouw de kleinste waarde (uit kolommen/locaties die we nog niet gehad hebben). Dus in de beginsituatie is een ondergrens voor de te verwachten kosten: $11 + 13 + 11 + 14 = 49$. Voor de deeloplossing a2b3 is die ondergrens $25 + 11 + 17 = 53$. In elke stap van het algoritme bekijken we de meest veelbelovende (= met de laagste ondergrens in dit geval) deeloplossing, breiden die op alle mogelijke manieren uit (zie boom hierna), berekenen de ondergrens 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 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 (ondergrens 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 gebouwen te koppelen aan locaties, te beginnen bij gebouw a. Eerst wordt de beginknoop uitgebreid op alle mogelijke manieren (4 stuks): a1, a2, a3, a4. Van de corresponderende knopen wordt de ondergrens bepaald (bijv. voor a1: $11 + 13 + 17 + 14 = 55$), en vervolgens wordt verdergegaan met de meest veelbelovende knoop, zijnde a2 in dit geval. Deze wordt op alle mogelijke manieren uitgebreid (levert 3 toelaatbare deeloplossingen), waarvoor vervolgens de ondergrenzen worden berekend. Ga door met de knoop met de kleinste ondergrens, in dit geval knoop b3. Deze kan op 2 manieren (toelaatbaar) worden uitgebreid; dit levert dan meteen twee oplossingen op, waarvan de beste kosten 64 heeft. Ten gevolge daarvan kan nu op 2 plekken gesnoeid worden (die met ondergrens ≥ 64), en er wordt verdergegaan met knoop a1. Wanneer de oplossing a1b3c4d2 is gevonden wordt de beste oplossing tot nu toe ge-update en worden alle resterende knopen behalve a3 gesnoeid. Etcetera.

c. In het algemeen kun je de waarde van een bekende oplossing gebruiken om veel eerder andere deeloplossingen weg te snoeien. Die hoeven dan niet meer onthouden te worden. Dit kan vooral voordelig zijn als oplossingen lang zijn, dus voor grote n in dit geval. Dan duurt het immers lang voordat je een oplossing hebt, en dus voordat je kan snoeien. In dit voorbeeld kun je meteen bij aanvang knoop a4 wegsnoeien.

Opgave 4.
a. Als $n = 1$: bord neerzetten op positie 1; opbrengst dan maximaal, nl. `opbrengst`[1]
Als $n = 2$: je kan alleen een bord op plek 1 óf plek 2 neerzetten; maximale opbrengst dus `maximum`(`opbrengst`[1], `opbrengst`[2]).
Als $n = 3$: je kan alleen een bord neerzetten op plek 1 óf op plek 2 óf op plek 3 (immers ze moeten > 1000 meter uit elkaar dus op plek 1 en plek 3 kan net niet allebei); maximale opbrengst dus `maximum`(`opbrengst`[1], `opbrengst`[2], `opbrengst`[3]).
b. Je kunt een reclamebord wel of niet op positie n neerzetten. Als je het bord wel op plek n neerzet kan er op plek $n-1$ en op plek $n-2$ geen bord staan. Het vorige reclamebord moet dus op positie $n-3$ of eerder: hetzelfde probleem, maar dan voor $n-3$. Plaats je het laatste reclamebord niet op plek n, dan reduceert het probleem tot hetzelfde probleem maar dan voor $n-1$. Recurrente formulering dus (met beginwaarden als in **a.**):

`maxtotaal`(n) = `maximum` (`maxtotaal`$(n-1)$, `maxtotaal`$(n-3)$ + `opbrengst`$[n]$).

c. Er is veel overlap tussen deelproblemen, waarbij veel berekeningen dus herhaald gebeuren. Bij de berekening van `maxtotaal`(10) wordt bijvoorbeeld `maxtotaal`(4) al zes keer berekend. Er valt veel voordeel te behalen als we een array gebruiken om tussenresultaten op te slaan: deeloplossingen worden hiermee maar één keer berekend! Bij dit probleem is het array een 1-dimensionaal array, want de tussenresultaten zijn de `maxtotaal`(i)-waarden.
Top down: we houden de recursieve vorm, maar zodra we een deeloplossing berekend hebben stoppen we die in het array. Bij elke recursieve aanroep kijken we vervolgens altijd eerst in het array of we het deelprobleem niet al opgelost hebben. In dat geval halen we de oplossing gewoon uit het array en doen we dus geen overbodige recursieve aanroep.
Bottom up: uit de recursieve formulering volgt een recurrente betrekking waaraan de array-entries (die de tussenresultaten/deeloplossingen bevatten) moeten voldoen. Die recurrente betrekking kun je ook van klein naar groot gebruiken om je array te vullen. Zie ook **d.** voor toepassing op het reclamebordprobleem.
c. Tussenresultaten zijn in dit geval de `maxtotaal`(i). Deze voldoen aan eenzelfde recursieve formulering als die uit **b.**, met dien verstande dat er overal i in plaats van n komt te staan. Noemen we het array van tussenresultaten even `totaal`, dan geldt dus:

`totaal`$[i]$ = `maximum`(`totaal`$[i-1]$, `totaal`$[i-3]$ + `opbrengst`$[i]$)
`totaal`$[i]$ = `opbrengst`[1] als $i = 1$
`totaal`$[i]$ = `maximum`(`opbrengst`[1], `opbrengst`[2]) als $1 = 2$
`totaal`$[i]$ = `maximum`(`opbrengst`[1], `opbrengst`[2], `opbrengst`[3]) als $i = 3$

Array-entry i heeft array-entries $i-1$ en $i-3$ nodig. Deze moeten dus bekend zijn. Het array moet derhalve van links naar rechts gevuld worden. Het bottom up dynamisch programmeren algoritme ziet er dan als volgt uit:

```
totaal[1] := opbrengst[1];
```

```
totaal[2] := maximum ( opbrengst[1], opbrengst[2] );
totaal[3] := maximum ( opbrengst[1], opbrengst[2], opbrengst[3]);
for i:=4 to n do
   if ( totaal[i-1] < totaal[i-3] + opbrengst[i] ) then
      totaal[i] := totaal[i-3] + opbrengst[i];
   else
      totaal[i] := totaal[i-1];
   fi
od
```

Opgave 5.
a. Verwissel $A[2]$ met $A[n-1]$, $A[4]$ met $A[n-3]$, etcetera, tot i halverwege het array is gekomen. Dus:

```
for ( i:= 2; i <= n/2; i+=2 )
   wissel(A[i], A[n-i+1]);
od
```

Dit algoritme doet $n/4$ verwisselingen als $n/2$ even is en $(n-2)/4$ als $n/2$ oneven is. Samengevat: $\lfloor n/4 \rfloor$ verwisselingen.
b. Idee: verwisselen van $A[2]$ met $A[n-1]$ brengt het probleem van 1 t/m n terug tot hetzelfde probleem, maar nu van 3 t/m $n-2$. Dat zijn 4 array-elementen minder.

```
hussel(i, j) ::   // eerste aanroep zal zijn met i=1 en j=n
   if ( j-i+1 >= 4 )
   // minstens 4 elementen
      wissel(A[i+1], A[j-1]);
      hussel{i+2, j-2);
   fi .
   // als 2 of 0 elementen: niets doen, het staat al goed
```

c. Divide and conquer: verdeel het array in een linkerhelft en een rechterhelft; zet de elementen links goed en rechts (recursief). Nu is de situatie als volgt: eerst $n/4$ oneven getallen, dan $n/4$ even getallen, vervolgens weer $n/4$ oneven getallen en daarna $n/4$ even getallen. We hoeven dus alleen de eerste $n/4$ even getallen te verwisselen met de tweede $n/4$ oneven getallen daar meteen achter. Voor de recursie: als het aantal betrokken elementen ($j-i+1$ dus) groter of gelijk aan 4 dan moeten we in twee delen splitsen, anders hoeven we niets te doen.

```
schuffel(i,j)::
   if (j-i+1 >= 4) then
      midden := (i+j) div 2; (*)
      schuffel(i, midden);
      schuffel(midden+1, j);
      half := (i+midden) div 2; (*)
      aantal := half-i+1;
      for k :=  half+1 to midden do
         wissel(A[k], A[k+aantal]);
      od
  fi .
```

(*) Met (i+j) div 2 wordt bedoeld: $\lfloor \frac{i+j}{2} \rfloor$ en (i+midden) div 2 betekent $\lfloor \frac{i+\texttt{midden}}{2} \rfloor$.