Uitgebreide uitwerking tentamen Algoritmiek
Dinsdag 2 juni 2009, 10.00 – 13.00 uur

Opgave 1.
a. Een toestand wordt bepaald door: het aantal lucifers op tafel, het aantal lucifers in het bezit van Romeo, het aantal lucifers in het bezit van Julia en degene die aan de beurt is. Merk op dat het ook al voldoende is om aan te geven of het aantal lucifers van Romeo/Julia even of oneven is.
In de begintoestand liggen er n lucifers op tafel, hebben Romeo en Julia er beiden 0 in hun bezit en is Romeo aan de beurt; in een eindsituatie is het aantal lucifers op tafel 0.
Een actie is het doen van een zet door degene die aan de beurt is: dus het wegnemen van één, twee of drie lucifers van de stapel in het midden van de tafel.
b.+ c. Zie het bijgevoegde bestand tad2009.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 Julia. Als Romeo in zijn eerste zet 3 lucifers wegneemt, moet Julia er vervolgens 1 weghalen en zal dan na Romeo's enig mogelijke zet winnen. In de andere twee gevallen neemt zij er 3 weg. Dan heeft ze ofwel meteen gewonnen, ofwel na de laatste, vastliggende, zet van Romeo.
d. Als Romeo in de eerste zet 3 lucifers weghaalt, resteert de toestand $(4, 3, 0)$ met Julia aan de beurt. Aangezien 3 oneven is, is dit dezelfde toestand als $(4, 1, 0)$ met Julia aan de beurt, en die is winnend voor Julia (zie 1. b,c). Als Romeo er in de eerste zet 2 wegneemt, resteert de toestand $(5, 2, 0)$, ofwel $(5, 0, 0)$ (2 is even), met Julia aan de beurt, en die is winnend voor Romeo. Dus als Romeo er 2 wegneemt in de eerste zet, zal hij bij perfect spel winnen. Dus het spel is winnend voor Romeo als $n = 7$. Voor de volledigheid: de toestand $(6, 1, 0)$ met Julia aan de beurt, resterend als Romeeo in de eerste zet 1 lucifer wegneemt, is winnend voor Julia. Als zij er 2 wegneemt zal ze winnen. Ga dit na.

Opgave 2.
a. Het initialiseren van de velden kan bijv. met een preordewandeling:

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

b. Recursieve formulering: als je in een knoop zit zet je de rechterbroer-velden en de extra-velden van zijn kinderen goed: wortel-->links->rechterbroer = `true`; wortel->links->extra = wortel->rechts; wortel->rechts->extra = wortel; wortel->links en wortel->rechts moeten dan wel beide $\neq$ `NULL` zijn. Andere gevallen analoog.

```
void broeders(knoop* wortel) {
  if (wortel != NULL) {
    // boom niet leeg (*)
    if (wortel->rechts != NULL) {
       wortel->rechts->extra = wortel;
```

```
        if (wortel->links != NULL) {
           wortel->links->rechterbroer = true;
           wortel->links->extra = wortel->rechts;
        }
     } // rechterkind bestaat
     else { // geen rechterkind
       if (wortel->links != NULL)
         wortel->links->extra = wortel;
     }
     broeders(wortel->links);
     broeder(wortel->rechts);
     // deze aanroepen mogen ook op plek (*)
   } // boom niet leeg
} // broeders
```

c. Tot de wortel naar boven lopen via de extra-pointers. Het aantal keer dat je echt omhoog gaat is dan het nivo van de knoop in de boom. De wortel is te herkennen aan het feit dat daar het exta-veld NULL is.

```
int nivo(knoop* wijzer) {
  knoop* loper = wijzer;
  int level = 0;
  while (loper->extra != NULL) {
    if (loper->rechterbroer)
      loper = loper->extra;
    loper=loper->extra;
    // je bent een nivo omhoog gegaan
    level++;
  }
} // nivo
```

Opgave 3.

a. Brute force (hier: uit de definitie): loop alle paren (i, j) af met $0 \leq i \leq j \leq n-1$, bereken de verschillen $A[j]-A[i]$ en retourneer de indices waarvoor dit verschil het grootst is. Het algoritme is dus $\Theta(n^2)$.

```
void dagen(int A[n], int& i, int& j) {
  int l, k; int max = 0;
  for (l=0; l<n; l++)
    for (k=l; k<n; k++) {
      if (A[k]-A[l] >= max) {
      // >= om te garanderen dat i en j een waarde krijgen
      // hiertoe kun je ook max op -1 initiaiseren en dan hier gewoon >
        i = l; j = k;
        max = A[k]-A[l];
      } // if
    }
} // dagen
```

b. In het geval dat i in de linkerhelft moet zitten en j in de rechterhelft, vinden we het maximale verschil simpelweg door in de rechterhelft de grootste waarde te vinden, en in de linkerhelft de kleinste waarde.

```
void simpel(int A[n], int& i, int& j) {
  int l; int helft = n/2;
  int min = A[0]; int max = A[helft];
  i = 0; j = helft;
  for (l=1; l<helft; l++) {
    if (A[l] < min) {                           n/2 - 1 vergelijkingen
      i = l; min = A[l];
    }
  } // for l links
  for (l=helft+1; l<n; l++) {
    if (A[l] > max) {                           n/2 - 1 vergelijkingen
      j = l; max = A[l];
    }
  } // for l rechts
} // simpel                            totaal:  n-1 vergelijkingen
```

c. Er zijn nu drie mogelijkheden: ofwel de gevraagde i en j zitten beide links (recursieve aanroep op de linkerhelft), ofwel ze zitten beide rechts (recursieve aanroep op de rechterhelft), ofwel i zit links en j zit rechts. Dit laatste geval is bekeken in 3.b. Merk op dat het basisgeval wordt gegeven door $r = l + 1$ aangezien $n \geq 2$, maar je mag ook als basisgeval $r = l$ nemen (en dan retourneer je $i = j = l$).

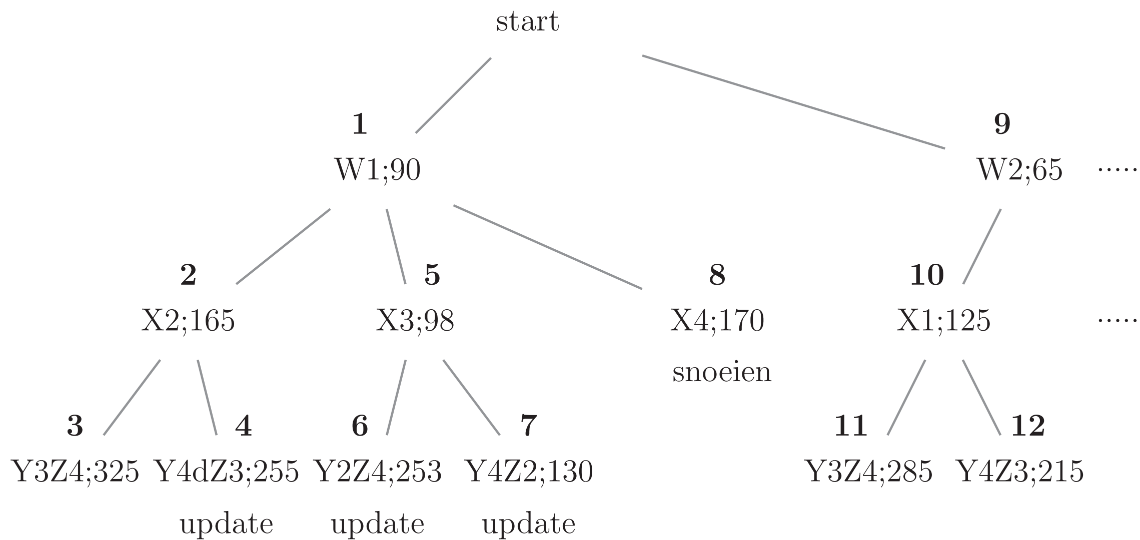
```
void maxwinst(int A[], int l, int r, int& i, int& j) {
  int il, jl, ir, jr, ii, jj, m;
  // het kan wel met minder variabelen
  if (r == l+1) {
    if (A[r]-A[l] >= 0) {
      i = l; j = r;
    }
    else {
      i = l; j = l;
      // of i=r;j=r; in elk geval op dezelfde dag kopen en verkopen
    }
  } // basisgeval twee elementen
  else {
  // meer dan twee elementen: recursieve aanroepen
    m = (l+r)/2;
    maxwinst(A, l, m, il, jl);
    maxwinst(A, m+1, r, ir, jr);
    if (A[jl]-A[il] > A[jr]-A[ir]) {
      i = il; j = jl;
    }
    else {
      i = ir; j = jr;
    }
```

```
    simpel(A, l, r, ii, jj);
    if (A[jj]-A[ii] > A[j]-A[i]) {
      j = jj; i = ii;
    }
  } // else recursieve aanroepen
} // maxwinst
```

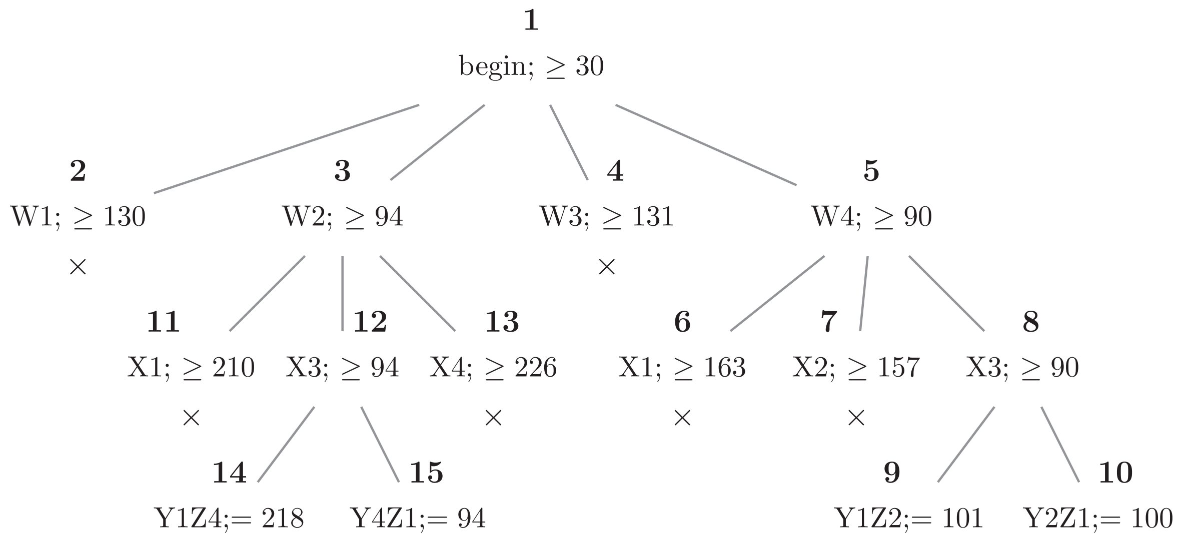
De functie `simpel` is hetzelfde als die uit b., met dien verstande dat l en r als parameters worden meegegeven en dat helft gelijk wordt aan $\lfloor (l + r)/2 \rfloor + 1$ (dat is in dit geval overigens gelijk aan $(l + r + 1)/2$).

Opgave 4.
a. Voorbeeld van een gretige strategie: kies combinaties (medewerker, taak) in volgorde van het aantal uren, decombinatie met het kleinste aantal uren eerst. Als een combinatie in strijd blijkt met de eisen (dus werknemer of taak was al eerder gekozen), neem dan de volgende. Kies dus steeds de kleinste dei nog kan; hopelijk leidt dit tot een minimale toewijzing. Echter, dit algoritme levert voor het voorbeeld de toewijzing W4X3Z1(*)Y2. (*): eerste keus was hier Y4, maar die kan niet omdat taak 4 al gekozen was. Deze heeft totaaltijd 100 uur, hetgeen niet optimaal is. Het voorgestelde gretige algoritme werkt dus niet. Merk op: bij een gretig algoritme zijn keuzes definitief; bij backtracking kunnen keuzes ongedaan gemaakt worden.
b. Genereer alle mogelijke toewijzingen stap voor stap door de medewerkers 1 voor 1 te koppelen aan een taak (in de volgorde 1 t/m 4 bijv.). We beginnen met persoon W, koppelen deze aan een taak, dan persoon X, dan persoon Y, etcetera. We proberen de werknemer dei aan de beurt is aan een taak te koppelen in de volgorde 1, 2, 3, 4. Een deeloplossing bestaat uit een deeltoewijzing van de eerste werknemers aan een unieke taak, en wel zo dat elke taak ook maar hooguit 1 keer voorkomt. Backtracking doet hier het volgende: een deeloplossing wordt uitgebreid door de volgende werknemer te koppelen aan taak 1. Als deze taak al voorkomt als taak bij een van de reeds gekoppelde personen, herzien we die keuze en proberen de volgende taak (enz.). Als een taak wel kan, gaan we de nieuwe deeltoewijzing op dezelfde manier uitbreiden. Als je alle taken bij een werknemer geprobeerd hebt ga je terug naar de vorige werknemer en probeer je daar de volgende taak (keuze bij vorige werknemer dus herzien). Verder houd je van je deeltoewijzing de tijd besteed aan het project bij en je onthoudt tevens de totaaltijd van de tot dusverre gevonden minimale volledige toewijzing. In elke stap vergelijk je de totale tijd van de deeltoewijzing met de tot nu toe minimale totaaltijd. Als de deeltijd groter of gelijk is aan de minimale 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 tijd indien nodig.

In de state-space-tree (zie volgende pagina) zijn de deeloplossingen die meteen herzien zijn omdat de taak al geweest is, zoals W1X1 en W1X2Y2, weggelaten om het plaatje overzichtelijk te houden. In dat geval is gebacktrackt omdat eenzelfde taak 2 keer voorkomt.
Het backtrackingalgoritme moet in dit geval bijna alle deeloplossingen bekijken. Er is op één plek gesnoeid omdat de tijd van de deeltoewijzing al groter was dan 170, de totaaltijd van de tot dan toe gevonden minimale toewijzing.



c. Een best-fit-first *branch and bound*-algoritme gebruikt een afschatting (hier een ondergrens!) op het verwachte totaal aantal uren 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.
Oplossingen (hier toewijzingen) worden net als bij backtracking stapsgewijs opgebouwd, maar bij uitbreiding van een knoop worden nu alle een-staps uitbreidingen daarvan gegeven, en van elk daarvan wordt de ondergrens bepaald. Bij backtracking bekijk je steeds maar één uitbreiding per stap. Bij branch and bound houd je meerdere deeloplossingen tegelijk bij (met hun ondergrens), bij backtracking maar één (en die breid je dan steeds verder uit). In elke stap kiezen we de knoop (deeloplossing) met de laagste ondergrens (best-fit-first), hopende op die manier zo snel mogelijk een zo klein mogelijke oplossing te vinden, waardoor je weer meer zult kunnen snoeien. In het algemeen zal branch and bound daarom sneller een minimale oplossing vinden, en meer snoeien, adn backtracking. Een mogelijk nadeel kan zijn dat meerdere deeloplossingen moeten worden bijgehouden (en waarvan steeds die met de kleinste ondergrens wordt gekozen). Hiervoor is een extra datastructuur nodig. Een ander mogelijk nadeel is de berekening van de ondergrens in elke knoop. Hoe ingewikkelder (en hoe beter?) die ondergrens, hoe meer tijd de berekening kost.
d. In dit geval nemen we bijvoorbeeld als ondergrens voor de te verwachten totale tijd: het totaal aantal uren van de betreffende deeloplossing + voor de andere personen (= andere nog te bekijken rijen) de kortst durende taak die we nog niet gehad hebben (kleinste waarde uit die rij waarvan de kolom/taak nog nietin de deeloplossing zit). Dus in de beginsituatie is een ondergrens voor het te verwachten aantal uren: $1 + 8 + 10 + 11 = 30$. Deze waarde correspondeert niet met een goede oplossing, maar geeft wel een ondergrens. Voor de deeloplossing W1X2 is die ondergrens $90 + 75 + 10 + 75 = 250$. Voor W3 is de ondergrens: $50 + 60 + 10 + 11 = 131$.
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.



Opmerking. De niet-toelaatbare deeloplossingen zoals W2X2 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 $\times$'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 werknemers te koppelen aan taken, te beginnen bij persoon W. Eerst wordt de beginknoop uitgebreid op alle mogelijke manieren (4 stuks): W1, W2, W3, W4. Van de corresponderende knopen wordt de ondergrens bepaald, en vervolgens wordt verdergegaan met de meest veelbelovende knoop, zijnde W4 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 X3. Deze kan op 2 manieren (toelaatbaar) worden uitgebreid; dit levert dan meteen twee oplossingen op, waarvan de beste kosten 100 heeft. Ten gevolge daarvan kan nu op 4 plekken gesnoeid worden (die met ondergrens $\geq 130, 131, 163, 157$), en er wordt verdergegaan met knoop W2. De uitbreidingen X1 en X4 kunnen meteen gesnoeid worden (want ondergrens ≥ 100). Alleen X3 wordt uitgebreid, en dit leidt tot de oplossing W2X3Y4Z1; de beste oplossing tot nu toe wordt ge-update en we hebben meteen de minimale oplossing gevonden, aangezien er geen kandidaatknopen meer over zijn.