Uitgebreide uitwerking tentamen Algoritmiek
Dinsdag 31 mei 2011, 10.00 – 13.00 uur

Opgave 1. **a.** Toestand: een collectie rechthoeken bestaande uit damstenen (of X-en), elk maximaal m bij n groot. Ook is van belang wie er aan de beurt is. Alternatief: een m bij n-bord waarbij elk vakje leeg is of een damsteen (X) bevat. De damstenen vormen rechthoeken. (In de begintoestand hebben we één rechthoek met $m * n$ damstenen, in de eindtoestand hebben we geen enkele steen meer over.)
Een actie is het weghalen van een rij uit één der rechthoeken door H als deze aan de beurt is, resp. het weghalen van een kolom uit één der rechthoeken door V als deze aan de beurt is. Tevens het omklappen van de beurt.
b. Zie het bijgevoegde bestand tad2011.pdf. Het spel is winnend voor V, die de eerste (of laatste) kolom moet weghalen en in zijn volgende zet de middelste kolom (zie in het plaatje de dubbele lijnen).
c. Uit de toestand-actie-ruimte uit **b.** zagen we al dat het geval 1 bij 3 winnend is voor V als die aan de beurt is. Hij neemt de middelste steen weg en wint in zijn volgende zet. Bekijk nu het geval 1 bij 5. Wil V winnen, dan moet hij in elk geval twee rijen overlaten (dus geen randkolom pakken). Als hij de tweede van links (of rechts, dat is symmetrisch) wegneem, blijven en een rij van 1 en een rij van 3 liggen. H kan maar één van de twee rijen weghalen. Er resteert dus X of XXX voor V. Beide gevallen zijn winnend voor V. Dus het geval 1 bij 5 is winnend voor V.
Nu het algemene geval. Om te winnen moet V de rij splitsen. Merk verder op dat als V een zet doet, er in totaal een even aantal stenen blijft liggen. V kan dus altijd zorgen dat hij splitst in twee oneven stapels. Hij kan dit bijvoorbeeld doen door de de tweede kolom van links weg te halen, waarbij er een rij van één steen en een rij van $n - 2$ stenen overblijven, beide van oneven lengte. H moet nu een van de twee rijen wegnemen, en er resteert weer een (kleinere) oneven rij met V aan de beurt. Als V deze strategie herhaalt zal hij uiteindelijk uitkomen op een rij van 1 steen, en wint.
Bonus. Als n even is kan V ofwel de rij met één korter maken (en vervolgens haalt H die in zijn geheel weg, en wint dus meteen), of hij splitst de rij in een even en een oneven rij. Immers als hij een kolom weghaalt blijven er $n - 1$ stenen liggen, en dat is oneven. Als H in dit geval de oneven rij in zijn geheel weghaalt blijft er een even rij liggen met V aan de beurt. Indien H deze strategie herhaalt komt V uiteindelijk uit op een 1 bij 2 rij, en dan verliest hij.

Opgave 2.
a. Het initialiseren van de reus-velden kan bijv. met een pre-ordewandeling:

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

b. Recursieve formulering algemeen (zonder te letten op `NULL`-pointers): je vindt het gevraagde maximum door de grootste te nemen van het maximum uit de linkersubboom,

het maximum uit de rechtersubboom en de waarde in de betreffende knoop zelf. In termen van de `reus`-velden:
`wortel->reus = grootste(wortel->links->reus, wortel->rechts->reus, wortel->info)`. (In plaats van `wortel->info` kun je `wortel->reus` zetten i.v.m. **a.**). Merk op dat `wortel->links->reus` en `wortel->rechts->reus` dus al bekend moeten zijn $\Longrightarrow$ postorde.

```
void reusachtig(knoop* wortel) {
  if (wortel != NULL) {
  // boom niet leeg
    reusachtig(wortel->links);
    reusachtig(wortel->rechts);
    if (wortel->links != NULL) {
      if (wortel->links->reus > wortel->reus)
        wortel->reus = wortel->links->reus;
      // grootste van linkersubboom en de wortel zelf
    }
    if (wortel->rechts != NULL) {
      if (wortel->rechts->reus > wortel->reus)
        wortel->reus = wortel->rechts->reus;
    }
  } // boom niet leeg
} //  reusachtig
```

c. Als `wortel->reus = wortel->info` is het maximum gevonden. Anders neem je de richting (links of rechts) waarvoor geldt dat het `reus`-veld gelijk is aan het `reus`-veld van de wortel. De grootste uit de boom is dan de grootste uit die betreffende subboom, dus daar moet je verder zoeken.

```
knoop* grootste(knoop* wortel) {
  knoop* loper = wortel;
  while (loper->info != loper->reus){
    if ( (loper->links != NULL) && (loper->links->reus == loper->reus) )
    // short-circuiting!
      loper = loper->links;
    else
      // het is hier niet nodig om op NULL te testen, want je weet hier zeker
      // dat de rechtersubboom niet leeg is: als de grootste waarde loper->reus
      // niet in de wortel zit zit en ook niet in de linkersubboom, moet hij
      // wel rechts zitten
      loper = loper->rechts;
  }
  return loper;
} // grootste
```

Opgave 3.
a. Als we twee negatieve (< 0) getallen bij elkaar optellen is het antwoord zeker < 0. Als we twee positieve (> 0) getallen bij elkaar optellen is het antwoord > 0. Beide gevallen

leveren dus nooit $= 0$ op. Ergo: van alle paren (i, j) met i en j beide oneven of i en j beide even weten we zeker dat $A[i] + A[j] \neq 0$.
b. Brute force: alle paren (i, j) met $i < j$ aflopen en testen of $A[i] + A[j] = 0$. Met inachtneming van de restrictie dat i en j niet beide even of beide oneven zijn.

```
int teller = 0;
for (i = 0; i < n; i++)
  for (j = i+1; j < n; j+=2)
  // nu worden bij elke even index i alleen oneven indices
  // j > i afgelopen, en analoog voor oneven i
    if (A[i] + A[j] == 0)
      teller++;
return teller;
```

c. We splitsen het array nu (herhaald, want recursie) in twee stukken. We tellen het aantal gevraagde paren in de linkerhelft (recursieve aanroep) en in de rechterhelft (recursieve aanroep). Vervolgens moeten we alleen nog paren indices (i, j) controleren waarbij i in de linkerhelft zit en j in de rechterhelft. Wederom onder de restrictie dat we alleen paren (i, j) bekijken met i even en j oneven, of met i oneven en j even.
Merk op dat het basisgeval wordt gegeven door `rechts` = `links` + 1 als we het stuk A[`links`], ..., A[`rechts`] bekijken. Eerste aanroep: aantal = aantal2(A,0,n-1);

```
int aantal2(int A[], int links, int rechts) {
  int teller = 0;
  int m, i, j;
  if (rechts == links+1) { // 2 elementen
    if (A[links]+A[rechts] == 0)
      return 1;
    else
      return 0;
  } // basisgeval twee elementen
  else {
  // meer dan twee elementen: recursieve aanroepen
    m = (links+rechts)/2; // m is altijd oneven
    // aantal paren links en aantal paren rechts optellen
    teller = aantal2(A, links, m) + aantal2(A, m+1, rechts);
    // en nu linkerhelft met rechterhelft vergelijken
    for (i = links; i < m; i++) { // even i links
      for (j = m+2; j <= rechts; j+=2)  // oneven j rechts
        if (A[i] + A[j] == 0)
          teller++;
    }
    for (i = links+1; i <= m; i++) { // oneven i links
      for (j = m+1; j < rechts; j+=2)  // even j rechts
        if (A[i] + A[j] == 0)
          teller++;
    }
    return teller;
  } // else meer dan 2 elementen
} // aantal2
```

Opgave 4.
a. Voorbeeld van een gretige strategie: begin in knoop a en neem de tak (a,..) met het grootste gewicht; dat geeft de volgende knoop op het pad. In dit geval is dat de tak (a,c). Ga vanuit c verder: kies de tak incident met c met zo groot mogelijk gewicht, en naar een knoop die nog niet geweest is. In dit geval is dat tak (c,e). Ga nu op dezelfde manier verder vanuit e. Dit levert de Hamiltonkring acebd op, met totaalgewicht 9+9+8+1+1=28. Dit is zeker niet maximaal, want de voorbeeldkring abdec heeft al een groter gewicht.
(Overigens kun je ook steeds de tak kiezen met het grootste gewicht, onder de voorwaarde dat je pas een kring hebt na het kiezen van de laatste (=vijfde) tak, en dat je een knoop die al twee keer gekozen is niet nogmaals kiest. Dat is een lastiger test, dus een minder handige gretige strategie. Hij levert overigens in dit geval dezelfde kring op als de eerdere strategie.)
b. (Deel)oplossingen worden *stap voor stap* opgebouwd. Een best-fit-first *branch and bound*-algoritme berekent voor de deeloplossingen (knopen in de state-space-tree) die bekeken worden een *afschatting (bound)* op de verwachte totale waarde (*) die je kunt krijgen als je de deeloplossing verder uitbreidt. Voor maximalisatieproblemen zoals hier wordt een bovengrens bepaald! Die bovengrens geeft dus aan dat elke legale uitbreiding van die deeloplossing een totaalwaarde $\leq$ die bovengrens heeft. ((*): de waarde die je wilt maximaliseren; in ons concrete probleem is dat het gewicht van een Hamiltonkring.)
De bovengrens wordt enerzijds gebruikt om 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 (*snoeien*). Als de huidige maximale waarde voor een reeds gevonden oplossing q bedraagt en de bovengrens in een knoop is $\leq q$, dan hoeft die deeloplossing/knoop niet verder bekeken te worden.
Oplossingen (hier Hamiltonkringen) worden stapsgewijs opgebouwd (conform de restricties, dus alleen legale/feasible uitbreidingen). Bij uitbreiding van een knoop (deeloplossing) worden altijd *alle een-staps uitbreidingen (branch)* 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: dit lijkt de meest veelbelovende knoop. Zodra we een nieuwe oplossing gevonden hebben *updaten* we uiteraard de best gevonden totale waarde.
c. We genereren hier (deel)oplossingen (Hamiltonkringen) door, beginnend bij knoop a, de betreffende deeloplossing een stap uit te breiden met alle mogelijke knopen die we nog niet eerder gehad hebben (restrictie). Verder maakt de richting waarin we een kring doorlopen niet uit (levert hetzelfde totaalgewicht), en dat betekent dat we kunnen eisen dat altijd b voor c komt. We breiden a dus uit met b, of d, of e. Voor al deze deeloplossingen (ab, ad, ae) bereken we de bovengrens (zie hieronder). Vervolgens kiezen we de deeloplossing met de grootste bovengrens (daar lijkt de kans het grootst om een goede oplossing te vinden), breiden deze weer op alle mogelijke manieren een stap uit en kiezen weer uit alle deeloplossingen degene met de hoogste bovengrens. Etcetera, zie de bijgevoegde state-space-tree.
Er zijn meerdere manieren om een (goede) bovengrens te vinden. We kiezen hier als bovengrens voor het te verwachten totale gewicht: voor elke knoop de twee grootste gewichten van aangrenzende takken bij elkaar opgeteld en het totaal gedeeld door twee, met dien verstande dat een tak (x,y) die in de deeloplossing gekozen is *moet* worden gekozen bij zowel knoop x als knoop y. In de beginsituatie is nog geen tak gekozen en wordt de bovengrens: $((7+9)+(7+8)+(9+9)+(6+4)+(9+8)):2 = 76:2 = 38$. Deze waarde correspondeert niet met een goede oplossing, maar geeft wel een bovengrens. Voor de deeloplossing ad is die bovengrens $((1+9)+(7+8)+(9+9+)+(1+6)+(8+9)):2 = 67:2 = 33,5$. Merk op

dat het gewicht van een Hamiltonkring altijd een geheel getal is, en als dat gewicht $\leq 33,5$ is, dan ook (geheel getal!) zeker ≤ 33. De bovengrens voor ad is dus 33. Nog een voorbeeld: voor aeb is de bovengrens: $\lfloor((5+9)+(8+7)+(9+9)+(4+6)+(5+9)) : 2\rfloor = \lfloor 71 : 2\rfloor = 35$. Uiteindelijk vinden we als maximale oplossing de Hamiltonkring abedc, met totaalgewicht 34. Zie verder de bijgevoegde state-space-tree in de file maxkring.pdf.

Opmerking. De niet-toelaatbare deeloplossingen zoals aa en abcb zijn voor de duidelijkheid niet in de boom opgenomen. Dat soort knopen wordt toch niet verder uitgebreid. De omcirkelde getallen bij de knopen geven de volgorde aan waarin de knopen worden *gekozen en een stap uitgebreid*. De $\times$'s geven aan dat de knoop niet verder hoeft te worden uitgebreid.

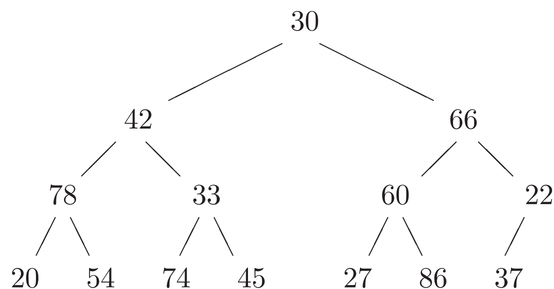
Opgave 5.

a. Een binaire boom heet een heap (maxheap) als:

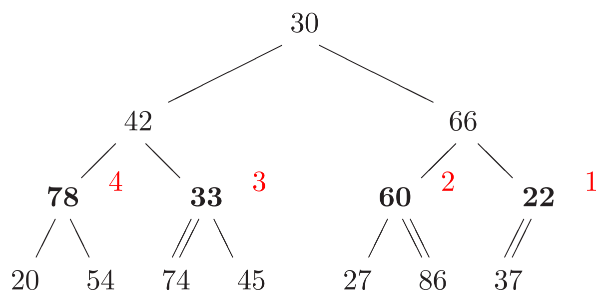
- de boom compleet is
- in *elke* knoop geldt dat de waarde in die knoop groter of gelijk is aan de waardes in de kinderen (heapeigenschap)

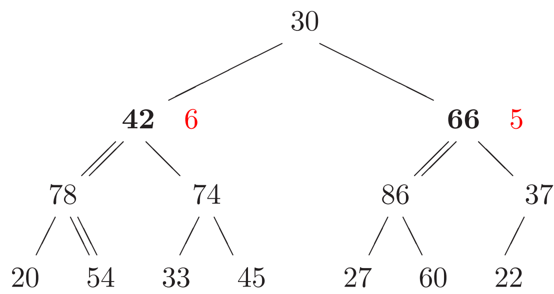
Een binaire boom is compleet als alle niveau's tot en met het een na onderste helemaal gevuld zijn, en als op het onderste niveau de knopen allemaal zover mogelijk links zitten.

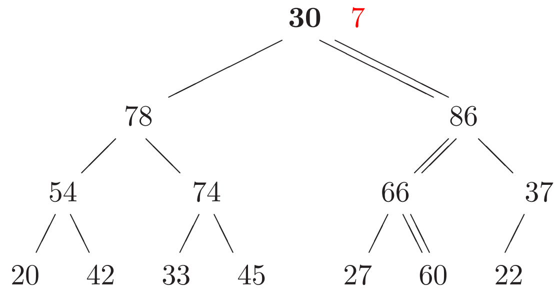
b. De complete binaire boom corresponderend met de gegeven rij getallen:

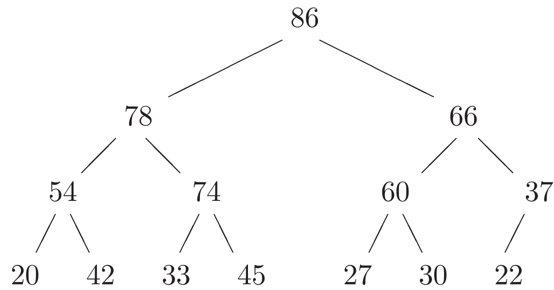


Heapify: van onder naar boven subbomen in hoopstructuur brengen; dat wil zeggen de heapeigenschap herstellen. De nummering geeft de volgorde aan waarin de subbomen "behandeld" worden. In hoopstructuur brengen gaat via herhaald verwisselen met het grootste kind. Bij de dubbele lijen wordt verwisseld met het grootste kind.

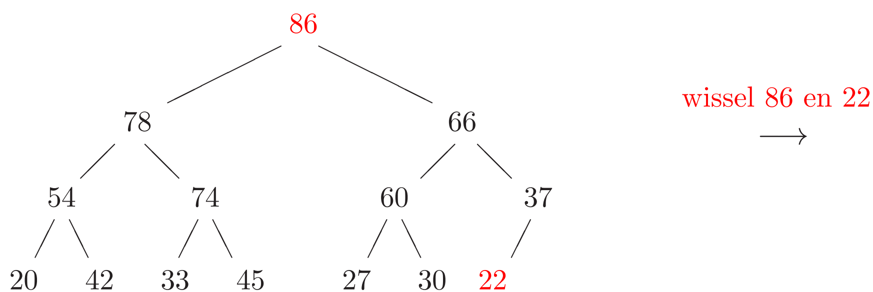


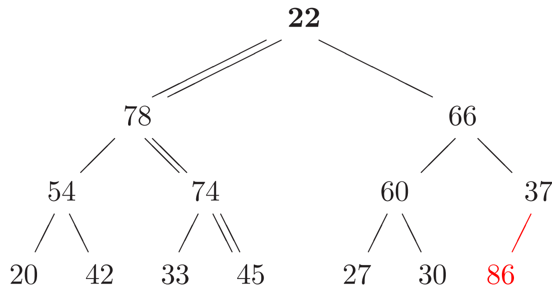




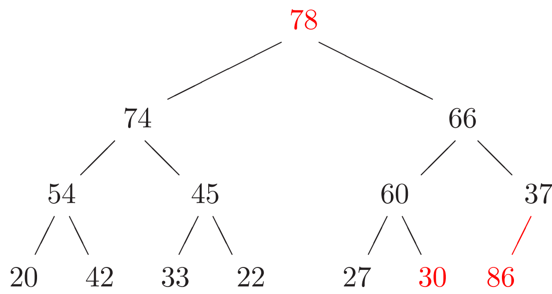


c. Sorteren met behulp van Heapsort werkt als volgt:
herhaald de grootste uit de wortel halen (en achteraan op zijn juiste plek neerzetten; dit komt neer op verwisselen met de laatste knoop uit de heap, die elke stap één knoop kleiner wordt) en vervolgens de resterende boom weer in heapstructuur te brengen (de heapeigenschap, die in de wortel verstoord is, herstellen) door herhaald verwisselen met het grootste kind. In onderstaande plaatjes stellen de zwarte knopen samen de (te herstellen) heap voor. Die wordt dus in elke stap één knoop kleiner.

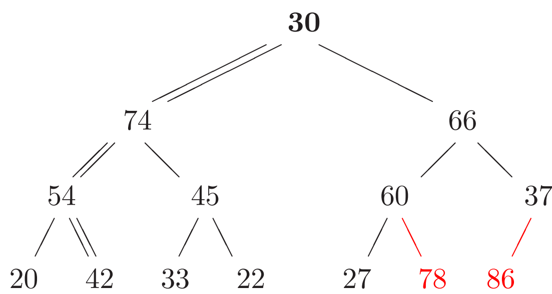




herhaald verwisselen met grootste kind ⟶



wissel 78 en 30 ⟶



herhaald verwisselen met grootste kind ⟶