Tentamen Algoritmiek
Dinsdag 7 juni 2016, 14.00 – 17.00 uur

Geef een **duidelijke toelichting** bij al je antwoorden.
Puntenverdeling: 1: 21; 2: 20; 3: 28; 4: 31. **Veel succes !**

Opgave 1. We bekijken een tweepersoonsspel dat wel bekend staat onder de naam *Aliquot*. Bij aanvang van het spel hebben we n lucifers, met n een geheel getal > 0. Er zijn twee spelers, in deze opgave Raemon en Kiki, die om de beurt een zet doen. Een zet is hier het weghalen van d lucifers, waarbij d een deler moet zijn van het aantal nog aanwezige lucifers m. Er moet gelden dat $d < m$. Het spel is afgelopen zodra een der spelers geen zet meer kan doen. Dit is het geval indien er nog maar 1 lucifer over is. In dat geval heeft degene die dan aan de beurt is verloren. We spreken af dat Kiki begint.

Voorbeeld van een mogelijk spelverloop voor $n = 44$. Mogelijke beginzetten voor Kiki zijn hier het weghalen van $22, 11, 4, 2$ of 1 lucifers. Zij kiest hier voor 22. Tussen haakjes staat steeds hoeveel lucifers door de aan de beurt zijnde speler worden weggenomen.

```
    K (22)    R (1)     K (3)     R (9)    K (3)    R (2)    K (2)    R (1)
 44 --->  22  --->  21  --->  18  --->  9  --->  6  --->  4  --->  2  --->  1
```

Raemon doet de laatste zet en laat Kiki zitten met 1 lucifer. Raemon heeft dus gewonnen.

a. (3 punten)
Wat zijn voor dit spel toestanden en acties (voor algemene n)?

b. (10 punten)
Teken de toestand-actie-ruimte voor het geval $n = 10$. Geef bij *elke* toestand in je toestand-actie-ruimte aan of deze winnend is voor K of voor R, te beginnen bij de eindstanden. Bepaal zo of het spel met $n = 10$ winnend is voor K of R en hoe gespeeld moet worden om te winnen. Kiki begint.
Toestanden die je al hebt uitgewerkt hoef je niet nogmaals volledig uit te werken. Zet daar wel bij wie er wint. Werk de standen die je na de eerste zet kunt krijgen uit van klein naar groot; in dat geval wordt je tekening niet te diep en blijft het overzichtelijk.

c. (8 punten)
Toon aan dat het spel
- winnend is voor de speler die begint indien n even is
- verliezend is voor de speler die begint (= winnend voor de tegenstander) indien n oneven is
Geef voor n even aan hoe gewonnen kan worden (dus een winnende strategie).

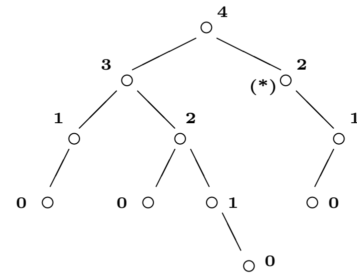
-vervolg op pagina 2

Opgave 2. Gegeven een binaire boom met ingang (ofwel: wortel) `wortel`. Hierin is `wortel` een pointer naar een `knoop`, die er als volgt uitziet:

```
class knoop {
  public:
    knoop* links;
    knoop* rechts;
    knoop* papa;
    int hoogte;
}; // knoop
```

Voorbeeld:

Bij de knopen staat de waarde van het hoogte-veld vermeld na het aanroepen van de functie uit **a**.



Bij aanvang van deze opgave hebben de `hoogte`-velden de waarde 0 en staan alle `papa`-pointers op NULL.

a. (10 punten)
Schrijf een *recursieve* C++-functie `void hoogte(knoop* wortel)` die in elke knoop het hoogte-veld vult met de hoogte van de binaire boom met de betreffende knoop als wortel. Tevens moet in elke knoop de `papa`-pointer naar de ouder van die knoop gaan wijzen (NULL voor de wortel). De hoogte van een boom is het grootste niveau dat voorkomt, waarbij de wortel op niveau nul zit. Merk op dat de boom leeg kan zijn.

We nemen nu aan dat de `hoogte`-velden en de `papa`-pointers reeds goedgezet zijn. We gaan een gegeven nieuwe knoop op een speciale manier toevoegen aan de boom. Zoals gebruikelijk voegen we deze ergens onderaan als blad toe. We zoeken de juiste plek door, beginnend in de wortel, naar de hoogte van de linker- resp. rechtersubboom te kijken. We lopen naar links als de hoogte van de linkersubboom kleiner of gelijk is aan de hoogte van de rechtersubboom, en anders naar rechts. Zo gaan we door totdat we de knoop kunnen toevoegen. In de voorbeeldboom zou de nieuwe knoop links onder knoop `(*)` komen.

b. (10 punten)
Schrijf een *niet recursieve* C++-functie `void toevoegen(knoop* wortel, knoop* nieuw)`, die de nieuwe knoop in de boom opbergt volgens bovenstaande methode. De pointer `nieuw` is een pointer naar de nieuwe knoop, waarvan de vier velden nog gevuld moeten worden.
Na het toevoegen van de knoop zijn mogelijk enige `hoogte`-velden niet meer correct. Gebruik de `papa`-pointers om de hoogtevelden –indien nodig– aan te passen.
Je mag in dit onderdeel aannemen dat de boom niet leeg is. De functie uit **a** mag niet aangeroepen worden.

-vervolg op pagina 3

Opgave 3. Gegeven een array $A = A[0], A[1], \ldots, A[n-1]$ dat n (≥ 4) gehele getallen bevat. Op de even posities staan positieve getallen en op de oneven posities negatieve getallen. Neem aan dat n een 2-macht is. Een *voorbeeld* met $n = 8$: $31, -85, 16, -42, 74, -28, 63, -59$.
De bedoeling is om het array door het verwisselen van elementen zo te reorganiseren dat alle negatieve getallen vooraan staan en de positieve achteraan. De onderlinge volgorde van de negatieve, resp. positieve getallen maakt niet uit. Het array uit het voorbeeld moet er na reorganisatie bijvoorbeeld zo uitzien: $-59, -85, -28, -42, 74, 16, 63, 31$.
We gaan dit probleem op drie manieren oplossen: iteratief, met decrease-and-conquer en met divide-and-conquer.

a. (6 punten)
Geef een eenvoudig *iteratief* algoritme dat het array A reorganiseert. Schrijf hiervoor een C++-functie `void reorganiseer1(int` A`[ ], int` n`)`. Gebruik twee lopende indices; de ene gaat van links naar rechts, de andere van rechts naar links.

b. (6 punten)
Geef een decrease-by-four algoritme voor bovenstaand probleem. Schrijf daartoe een *recursieve* C++-functie `void reorganiseer2(int` A`[ ], int` i`, int` j`)`. Deze functie moet het (deel)array $A[i], A[i+1], \ldots, A[j]$, met $j - i + 1$ een viervoud[1] reorganiseren. De aanroep `reorganiseer2(`$A, 0, n-1$`);` levert dan uiteraard het hele array gereorganiseerd op.

c. (10 punten)
Geef nu een divide-and-conquer algoritme dat het probleem oplost. Het array dient hiervoor in twee gelijke delen te worden verdeeld. Schrijf daartoe een *recursieve* C++-functie
`void reorganiseer3(int` A`[ ], int links, int rechts)` die het probleem oplost voor het deelarray A[`links`], ..., A[`rechts`] ter lengte een 2-macht.

d. (6 punten)
(i) Hoeveel verwisselingen doet het iteratieve algoritme uit **a** voor $n = 4, 8, 16$? En voor algemene $n \geq 4$ (een 2-macht)?
(ii) Hoeveel verwisselingen doet het recursieve algoritme uit **c** voor $n = 4, 8, 16$?

-vervolg op pagina 4

[1] het aantal elementen is dus een viervoud

Opgave 4. De gemeente Leiden is van plan om n verschillende gebouwen neer te zetten, en heeft daartoe n verschillende locaties, genummerd 1 t/m n, aangewezen. Op elke locatie komt precies één gebouw. De kosten van het bouwen hangen uiteraard af van het soort gebouw en van de locatie. Deze kosten zijn op voorhand bekend en opgeslagen in een n bij n tabel. Uiteraard wil de gemeente de totale kosten van de gebouwen minimaliseren. Er wordt dus een *toewijzing* gezocht van gebouwen aan locaties, met *minimale* totale kosten.

Voorbeeld met $n = 4$:

	locatie 1	locatie 2	locatie 3	locatie 4
gebouw A	9	4	3	2
gebouw B	5	3	1	4
gebouw C	4	5	3	2
gebouw D	5	6	2	6

De toewijzing A3, B1, C2, D4 betekent dat gebouw A op locatie 3 wordt gebouwd, gebouw B op locatie 1, gebouw C op locatie 2 en gebouw D op locatie 4. De totale kosten van deze toewijzing zijn $3 + 5 + 5 + 6 = 19$ miljoen euro. Dit is niet minimaal.

Bij branch and bound worden oplossingen (dus toewijzingen) component voor component opgebouwd. In dit geval gaan we dit doen door de gebouwen in alfabetische volgorde (eerst A, dan B, dan ...) een voor een te koppelen aan locaties (1, 2, ...). Dit geeft deeloplossingen zoals A3, B1 die we kunnen uitbreiden door vervolgens C aan een locatie te koppelen (waarna de toewijzing overigens vastligt). **Genereer bij c je oplossingen zoals hierboven beschreven!**

a. (10 punten)
Leg uit hoe best-fit-first *branch and bound* werkt voor minimalisatieproblemen in het algemeen. Geef daarbij o.a. aan wat met branch bedoeld wordt en wat met bound, wat best-fit-first betekent, wanneer gesnoeid wordt, enz.

b. (6 punten)
Geef twee belangrijke verschillen aan tussen backtracking en branch and bound en leg uit waarom branch and bound (meestal) sneller een minimale oplossing zal vinden.

c. (10 punten)
Pas de methode best-fit-first branch and bound toe op het voorbeeld en teken de bijbehorende state-space-tree (toestandsboom). Geef daarin aan in welke volgorde de knopen bekeken worden en welke deeloplossingen gesnoeid worden en waarom. Leg ook duidelijk uit welke afschatting je bij deeloplossingen gebruikt voor de uiteindelijke totale kosten. Bereken de afschattingen via de rijen. Geef voor ten minste drie illustratieve gevallen de berekening van die afschatting.

d. (5 punten)
Formuleer een *gretig* algoritme voor het probleem, pas dit toe op het voorbeeld en geef de aldus gevonden oplossing.