Tentamen Algoritmiek
Dinsdag 5 juni 2018, 14.00 – 17.00 uur

Als er om uitleg, toelichting of motivatie gevraagd wordt bij een opgave, is het belangrijk om die ook te geven.
Als je het antwoord op een onderdeel niet weet, en je hebt dat antwoord nodig bij een later onderdeel, dan kun je het antwoord 'kopen' bij de docent.
Globale puntenverdeling: 1: 33 pt; 2: 24 pt; 3: 13 pt; 4: 30 pt. **Veel succes!**

---

1. Het handelsreizigersprobleem (Travelling Salesman Problem) luidt: gegeven een complete, ongerichte graaf met n knopen en met gewichten op de takken. Geef een Hamiltonkring met minimaal totaalgewicht.

   (a) We zijn dit probleem meerdere keren tegengekomen bij de colleges. Bij het zoeken naar de optimale oplossing construeerden we zelfs bij exhaustive search niet alle mogelijke Hamiltonkringen door de graaf.

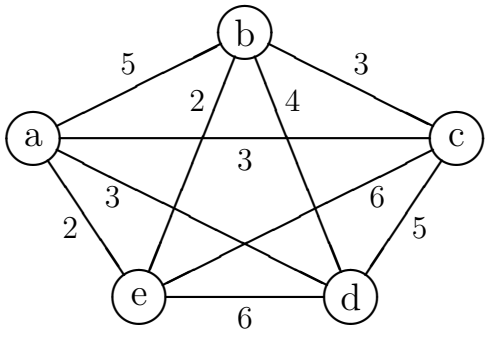
   Leg uit op welke manier(en) we zelfs bij exhaustive search het aantal kringen dat we construeerden beperkten. Leg ook uit waarom we dit konden doen zonder het gevaar te lopen dat we de optimale oplossing niet meer zouden vinden.

   (b) Leg uit hoe best-first branch-and-bound werkt voor minimalisatieproblemen in het algemeen. Geef daarbij o.a. aan hoe (deel-)oplossingen gegenereerd worden, wat met branch bedoeld wordt en wat met bound, wat best-first betekent, wanneer gesnoeid wordt, enz.

   (c) Tijdens het college hebben we een branch-and-bound algoritme voor het handelsreizigersprobleem behandeld. Maken we daarbij gebruik van een ondergrens of een bovengrens?

   En hoe berekenen we deze grens voor de eerste toestand in het algoritme (als we nog geen tak gekozen hebben). Beschrijf de berekening voor een algemeen handelsreizigersprobleem met $n \geq 3$ knopen.

   N.B.: in dit onderdeel en het volgend onderdeel is het de bedoeling dat je de grens **niet** door 2 deelt.



   (d) Pas het branch-and-bound algoritme met de grens uit het vorige onderdeel toe op de graaf hierboven en teken de bijbehorende state-space-tree, met bij elke knoop (deeloplossing) de relevante informatie. Geef ook aan in welke volgorde de knopen zijn aangemaakt, welke knopen gesnoeid worden en waarom.

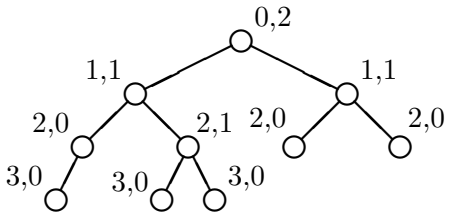
   Wat is dus de optimale oplossing?

---

2. Deze opgave gaat over binaire bomen, bestaande uit knopen uit de klasse `knoop` die er als volgt uitziet:

```
class knoop
{ public:
    knoop* links;
    knoop* rechts;
    knoop* ouder;
    int niveau;
    int nullafstand;
}; // knoop
```

Voorbeeld:
Bij de knopen staat de waarde van de velden `niveau` en `nullafstand` vermeld ná het aanroepen van de functie uit onderdeel (a), respectievelijk (b).



Bij aanvang hebben alleen de velden `links` en `rechts` in de boom een zinvolle waarde. De andere velden gaan gevuld worden bij onderdelen (a) en (b).

(a) Schrijf een *recursieve* C++-functie `void vulouderniveau (knoop *w, knoop *oud, int niv)` die in elke knoop in de (sub)boom met wortel `w` het `ouder` veld en het `niveau` veld vult met de juiste waarde.

Het `ouder` veld moet voor elke knoop (inderdaad) gaan wijzen naar de ouder in de boom. Het `niveau` veld moet het niveau van de knoop in de boom (de afstand vanaf de wortel) worden.

Je mag aannemen dat de parameter `oud` bij aanroep van de functie de ouder van knoop `w` is (of NULL voor de wortel van de boom), en dat de parameter `niv` het correcte niveau is voor knoop `w`.

(b) Zoals het niveau van een knoop de afstand vanaf de wortel is, willen we het `nullafstand` veld zien als de kortste afstand naar een NULL-pointer in de boom, lopend via de `links` en `rechts` pointers.

Zo heeft een knoop die nul of maar één kind heeft, `nullafstand` = 0. Immers, minstens één van zijn velden `links` en `rechts` is gelijk aan NULL. Een knoop met twee kinderen, waarvan één kind een blad is, heeft `nullafstand` = 1. Enzovoort, zie de voorbeeldboom hierboven.

Schrijf een *recursieve* C++-functie `void vulnullafstand (knoop *w)` die in elke knoop in de (sub)boom met wortel `w` het `nullafstand` veld vult met de juiste waarde.

(c) Ga er nu vanuit dat van alle knopen in een binaire boom de vijf velden correcte waardes hebben.

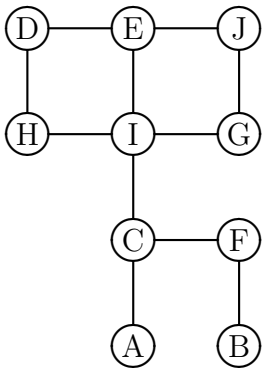
Het `nullafstand` veld kunnen we gebruiken bij het toevoegen van knopen aan de boom, zo dicht mogelijk bij de wortel. Schrijf daartoe een *niet-recursieve* functie `void voegtoe (knoop *w, knoop *nieuw)`, die knoop `nieuw` toevoegt aan een boom met wortel `w`, zo dicht mogelijk bij deze wortel.

Bij aanroep van `voegtoe` zijn de velden `links` en `rechts` van `nieuw` al gelijk aan NULL. De andere velden moeten nog goed gezet worden. Na afloop moeten ook van alle andere knopen in de boom de velden nog correcte waardes hebben. Maar nogmaals: de functie moet niet-recursief zijn, en mag ook geen recursieve hulpfuncties gebruiken.

Je mag er bij dit onderdeel vanuit gaan dat de parameters `w` en `nieuw` niet NULL zijn.

---

3. (a) Beschouw de volgende ongerichte graaf G:



Maak een DFS (depth-first search) wandeling door G, startend in knoop A. Wanneer een knoop meerdere buren heeft, handel die dan in alfabetische volgorde af.

Geef de resulterende DFS-boom. Geef daarin aan in welke volgorde de knopen voor de eerste keer worden bereikt (en op de stapel gezet), en in welke volgorde ze helemaal zijn afgehandeld (van de stapel worden gehaald).

(b) Om een DFS-wandeling door een graaf G te maken, kunnen we de volgende pseudo-code gebruiken.

```
ALGORITME DFS (G)
    // Implementeert DFS-wandeling door gegeven graaf G
    // Invoer: Graaf G
    // Uitvoer: Graaf G met zijn knopen genummerd in de volgorde
    //          waarin ze bij DFS-wandeling voor het eerst worden bereikt
{
    for  (elke knoop v in G)  do
        mark[v] = 0;   // nog niet bezocht
    od
    teller = 0;
    for  (elke knoop v in G)  do
        if  (mark[v] == 0)  then
            dfs (v);
        fi
    od
}
```

Hierin is `teller` een globale teller die gebruikt kan worden om de knopen te nummeren in de volgorde waarin ze bij de DFS-wandeling voor het eerst worden bereikt. Verder is `mark` een globaal array, waarin de nummering wordt opgeslagen.

Geef een *recursieve* pseudo-code implementatie van de functie `dfs (v)`. De functie moet alle nog onbezochte knopen die via een pad met `v` verbonden zijn, bezoeken, en ze in het array `mark` nummeren in de volgorde waarin ze worden bereikt. Je hoeft hierbij niet te kiezen tussen een adjacency matrix of een adjacency list representatie.

---

4. Langs een lange weg liggen achtereenvolgens de steden $0, 1, 2, \ldots, n$. Voor elke i en j met $i < j$ is er een bus van stad i naar stad j, met een bijbehorende prijs prijs$[i][j]$. Voor de volledigheid definiëren we prijs$[j][j] = 0$ voor elke j.

   We willen een busreis maken van stad 0 naar stad n met de goedkoopste combinatie van buskaartjes. Laat voor $j = 0, 1, 2, \ldots, n$, kosten(j) de kosten zijn voor de goedkoopste busreis van stad 0 naar stad j.

   (a) Beredeneer dat kosten(j) voldoet aan de volgende recurrente betrekking:

$$\text{kosten}(j) = \begin{cases} 0 & \text{als } j = 0 \\ \min_{0 \le i < j}(\text{kosten}(i) + \text{prijs}[i][j]) & \text{als } j \ge 1 \end{cases}$$

   (b) Een rechtstreekse vertaling van deze recurrente betrekking in een recursief algoritme kan de volgende C++-functie opleveren (afgezien van declaratie van variabelen):

```
int kosten (int j)
{
  if (j == 0) then
    return 0;
  else
  { temp = prijs[0][j];  // i = 0
    for (i=1; i<j; i++)
    { hulp = kosten(i) + prijs[i][j];
      if (hulp < temp)
        temp = hulp;
    }
    return temp;
  }
}
```

   Stel dat we met deze functie kosten(n) willen berekenen. Dan kunnen we ons afvragen hoeveel (recursieve) aanroepen kosten(j) er in totaal worden gedaan, voor $j = 0, 1, 2, \ldots, n$ bij elkaar. Geef het resulterende totaal aantal (recursieve) aanroepen voor $n = 0$, voor $n = 1$, voor $n = 2$, voor $n = 3$ en voor $n = 4$. Laat ook zien hoe je aan je antwoorden komt, bijvoorbeeld met bomen van (recursieve) aanroepen.

   (c) Bij dynamisch programmeren slaan we alle waardes kosten(j) op in een tabel (array) kosten. We noemen het dan kosten$[j]$ (met blokhaken). Geef een algoritme (in C++ of pseudo-code) dat kosten$[n]$ berekent met **bottom-up** dynamisch programmeren.

   (d) Pas het dynamisch programmeren algoritme van het vorige onderdeel toe op de tabel met prijzen prijs$[i][j]$ hiernaast.
   Laat voor iedere waarde van j (dus voor $j = 0, 1, 2, 3, 4$) zien hoe je de waarde kosten$[j]$ berekent.

	j				
i	0	1	2	3	4
0	0	12	18	24	31
1	-	0	5	13	20
2	-	-	0	7	13
3	-	-	-	0	8
4	-	-	-	-	0

   (e) Wat is de tijdcomplexiteit van het algoritme van onderdeel (c) om kosten$[n]$ te berekenen? Motiveer je antwoord door een basisoperatie in het algoritme aan te wijzen en te bepalen hoe vaak deze operatie wordt uitgevoerd.