Tentamen Algoritmiek
Donderdag 6 juni 2019, 14.00 – 17.00 uur

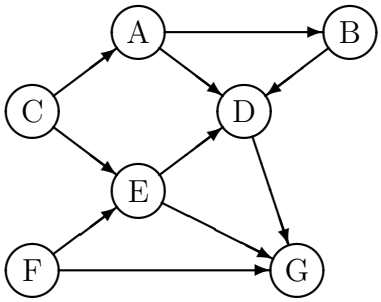
Als er om uitleg, toelichting of motivatie gevraagd wordt bij een opgave, is het belangrijk om die ook te geven.
Als je het antwoord op een onderdeel niet weet, en je hebt dat antwoord nodig bij een later onderdeel, dan kun je het antwoord 'kopen' bij de docent.
Globale puntenverdeling: 1: 21 pt; 2: 25 pt; 3: 26 pt; 4: 28 pt. **Veel succes!**

---

1. (a) In een gerichte acyclische graaf (*directed acyclic graph = DAG*) kun je een *topologische ordening* van de knopen bepalen. Wat verstaan we onder zo'n topologische ordening?

   (b) Beschrijf (in woorden of met (pseudo-)code, maar in ieder geval duidelijk en volledig) een *decrease-by-one-and-conquer* algoritme om in een DAG een topologische ordening te bepalen.

   (c) Pas het algoritme uit onderdeel (b) toe op onderstaande DAG. Geef voor elke stap van het algoritme duidelijk aan, waartussen je moet kiezen en wat je vervolgens doet, bijvoorbeeld in een overzichtelijke tabel. Licht toe hoe je uitwerking gelezen moet worden.



   Wat is de resulterende topologische ordening?

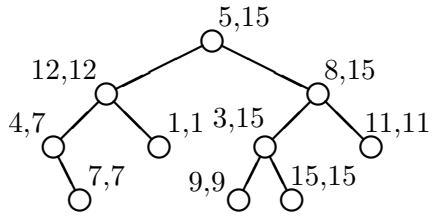
   (d) Beredeneer waarom er in een gerichte **cyclische** graaf (een graaf **met** één of meer kringen dus) geen topologische ordening kan bestaan.

---

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* oud;
    int info;
    int maxim;
}; // knoop
```

Voorbeeld:
Bij de knopen staan de waardes van de velden `info` en `maxim` vermeld ná het aanroepen van de functie uit onderdeel (b).



Bij aanvang hebben de velden `info` in de knopen al een waarde. De velden `maxim` en `oud` hebben op dat moment nog geen zinvolle waarde. Je mag er dus zelfs niet van uitgaan dat ze op 0 of NULL geïnitialiseerd zijn. Deze twee velden gaan gevuld worden bij onderdelen (a) en (b).

(a) Het veld `oud` in een knoop is bedoeld voor een pointer naar de ouder van die knoop in de boom. Voor de wortel van de boom moet het veld NULL worden.

Schrijf een *recursieve* C++-functie `void vuloud (knoop *w)` die de velden `oud` in de boom (of subboom) met wortel `w` vult met de juiste waarde.

(b) Het veld `maxim` in een knoop is bedoeld voor de maximale `info`-waarde in de subboom met die knoop als wortel. Voor een voorbeeld, zie de boom hierboven.

Schrijf een *recursieve* C++-functie `void vulmaxim (knoop *w)` die de velden `maxim` in de boom (of subboom) met wortel `w` vult met de juiste waarde.

(c) Ga er nu vanuit dat de boom minstens twee knopen bevat, dat alle velden `info` verschillend zijn, en dat de velden `oud` en `maxim` gevuld zijn met de juiste waardes. Schrijf een **niet-recursieve** functie `int verwijdermax (knoop *w)`, die de maximale `info`-waarde uit de boom met wortel `w` verwijdert en die waarde retourneert. Na afloop moeten de velden `oud` en `maxim` in de resterende boom nog steeds gevuld zijn met de juiste waarde.

Je moet dus op zoek gaan naar de knoop met de maximale waarde, die verwijderen, en er vervolgens voor zorgen dat alle velden in de resterende boom weer gevuld zijn met de juiste waarde.

Kies één van de volgende twee varianten van dit onderdeel:

i. Ga ervanuit dat, net als in de voorbeeldboom, de knoop met de maximale waarde een blad is, die je dus eenvoudig uit de boom kunt verwijderen.

ii. (5 punten meer dan de vorige variant:) Maak de functie `verwijdermax` algemeen, zodat ze ook werkt als de knoop met de maximale waarde geen blad is. In dat geval vervang je de maximale waarde door de waarde van een (zelf te kiezen) blad in zijn subboom, en verwijder je vervolgens dat blad.

N.B.: Dit kan een lastig onderdeel zijn. Besteed er, als het niet lukt, niet te veel tijd aan.

---

3. Bij het toewijzingsprobleem (*assignment problem*) hebben we n personen en n jobs. Elke persoon moet één job uitvoeren. Wanneer persoon i job j uitvoert, kost dat $kosten[i][j]$. Doel van het toewijzingsprobleem is om een zodanige toewijzing van jobs aan personen te krijgen, dat de totale kosten worden geminimaliseerd. We willen het toewijzingsprobleem oplossen met behulp van *best-first branch-and-bound*.

   (a) Maken we bij branch-and-bound voor dit probleem gebruik van een ondergrens of van een bovengrens? Motiveer je antwoord.

   (b) Beschrijf een geschikte ondergrens of bovengrens (afhankelijk van je antwoord bij (a)) voor het toewijzingsprobleem. Geef zo'n beschrijving zowel voor de begintoestand (als er nog helemaal geen jobs aan personen zijn toegewezen) als voor een algemene deeloplossing. Ga ervanuit dat we per rij werken.

   (c) Pas de methode best-first branch-and-bound toe op de volgende kostenmatrix met $n = 4$:

	job 1	job 2	job 3	job 4
Alexander	4	5	3	8
Beatrix	5	9	5	8
Claus	5	6	4	8
Donald	9	4	5	5

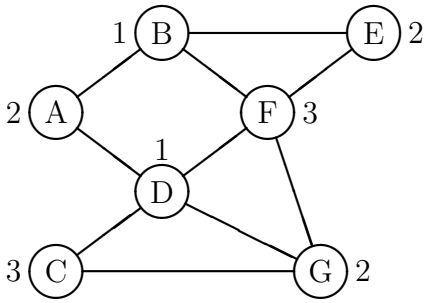
   Teken de bijbehorende *state-space-tree*, met bij elke knoop (deeloplossing) de relevante informatie (waaronder ook de opbouw van de ondergrens of bovengrens). Geef ook aan in welke volgorde de knopen zijn aangemaakt, welke knopen gesnoeid worden en waarom.

   Wat is dus de optimale oplossing?

   (d) Hoe ziet de state-space-tree bij best-first branch-and-bound voor een toewijzingsprobleem met $n = 4$ er in het slechtste geval (d.w.z.: met zoveel mogelijk toestanden) uit? Je hoeft geen getallen in te vullen in de knopen; het gaat om om de vorm van de boom. Hoeveel knopen bevat de state-space-tree in dat geval?

---

4. Laat G een ongerichte graaf zijn. Een *kleuring* van G met K kleuren, is een kleuring van de knopen van G (elke knoop krijgt een van de K kleuren), waarbij voor elke tak (i, j) in de graaf geldt dat knopen i en j verschillende kleuren krijgen. Een voorbeeld van een kleuring van een graaf met drie kleuren is:



   De nummers bij de knopen stellen de kleuren 1, 2 en 3 voor. We kunnen ons nu afvragen wat het **minimale** aantal kleuren is waarmee we een gegeven graaf kunnen kleuren.

Z.O.Z.

(a) Een mogelijk gretig algoritme voor het bepalen van het minimale aantal kleuren werkt als volgt:

```
kleur de eerste knoop met de eerste kleur;
for (alle andere knopen i in de graaf)
{ kleur knoop i met de eerst mogelijke kleur
    (zonodig met een nieuwe kleur);
}
```

i. Pas dit gretige algoritme toe op de voorbeeldgraaf aan het begin van deze opgave. Loop de knopen daarbij in de for-lus in alfabetische volgorde af. Vermeld stap voor stap welke knoop welke kleur krijgt en waarom.

ii. Wat is de worst case tijdcomplexiteit van het gretige algoritme, als functie van het aantal knopen N? Motiveer je antwoord. Laat in je motivatie ook zien hoe je de instructie `kleur knoop i met de eerst mogelijke kleur;` precies uitvoert. Ga ervanuit dat de graaf wordt gerepresenteerd door een adjacency matrix `int graaf[N][N]`.

(b) Het gretige algoritme geeft niet altijd een optimale oplossing. We gaan daarom nu backtracking gebruiken om het minimale aantal kleuren te bepalen waarmee een graaf gekleurd kan worden. Ga er nu vanuit

- dat de N knopen in de graaf genummerd zijn als $0, 1, 2, \ldots, N-1$,
- dat de kleuren genummerd zijn als $1, 2, 3, \ldots$,
- en dat de graaf gerepresenteerd is door een adjacencymatrix `int graaf[N][N]` (een globale variabele).

Schrijf een *recursieve* C++-functie `void bepaalminkleuren (int N, int kleuring[], int i, int K, int &minkleuren)`, die met behulp van backtracking het minimale aantal kleuren bepaalt om de graaf te kleuren. De parameters `kleuring`, i en K betekenen

- dat we tot nu toe kleuren `kleuring[0]`, ..., `kleuring[i-1]` bepaald hebben voor de eerste i knopen, waarbij we kleuren $1, 2, \ldots, K$ daadwerkelijk gebruikt hebben,
- en dat we in deze recursieve aanroep mogelijke waardes voor `kleuring[i]` gaan proberen.

De parameter `minkleuren` bevat het minimale aantal kleuren om de totale graaf te kleuren, dat we tot nu toe gevonden hebben.

De eerste aanroep zal van de vorm `bepaalminkleuren (N, kleuring, 0, 0, minkleuren);` zijn, waarbij `minkleuren` is geïnitialiseerd op `INT_MAX`.[1]

---

[1] Inderdaad, we hebben aan het begin nog $K = 0$ kleuren gebruikt.