ALGORITMIEK: antwoorden werkcollege 6

opgave 1.
Genereer de magische vierkanten stap voor stap: vul de vakjes (i, j) van het vierkant (bijvoorbeeld rij voor rij en per rij van links naar rechts) met de getallen 1 t/m n^2 (een voor een). Controleer of de betreffende waarde niet al eerder (in een eerdere rij/kolom) voorkwam (vergelijk permutaties genereren van college). Zo ja, dan volgende getal proberen, zo nee dan controleren of de tot dusver verkregen rijsom (rij i)/kolomsom (kolom j) niet al te groot is, of te klein (*). Als (i, j) op een hoofddiagonaal ligt, voer dan een dergelijke controle uit voor de diagonaalsom.
Als een van de tot dusver verkregen sommen te groot of te klein is, dan heeft het geen zin om verder te gaan. Probeer dan op plek (i, j) de volgende waarde. Als de sommen nog OK zijn, breid dan de deeloplossing op dezelfde manier verder uit. Als alle waarden op een plek geprobeerd zijn moet de waarde van de vorige positie worden herzien.
(*) Te groot betekent bijv. dat de huidige som al groter dan $\frac{n(n^2+1)}{2}$ is, maar kan ook wat subtieler. Je kunt, gegeven het aantal vakjes dat in de betreffende rij/kolom/diagonaal nog gevuld moet worden, een afschatting maken van hetgeen ten minste de rijsom/kolomsom/diagonaalsom zal worden. Overigens kun je ook op soortgelijke manier een afschatting geven van de maximaal te bereiken rijsom/kolomsom/diagonaalsom, en daaruit kun je mogelijk concluderen dat je de waarde $\frac{n(n^2+1)}{2}$ niet meer kan bereiken. Dat is ook een reden om niet verder te gaan met uitbreiden.

opgave 2.
a. Er zijn n knopen; elke knoop kan met m mogelijke kleuren gekleurd worden; dus er zijn in principe maximaal m^n mogelijke kleuringen. Daarvan zullen er in het algemeen een heleboel niet aan de restrictie (buren hebben verschillende kleuren) voldoen. Echter voor de graaf die n knopen heeft en geen takken zijn al deze kleuringen goed. Overigens is het minimale aantal kleuren nodig om zo'n graaf te kleuren gelijk aan 1.
b. De complete graaf met n knopen bevat per definitie tussen elk tweetal knopen een tak: elke knoop is dus verbonden met alle $n - 1$ andere. Ergo: deze graaf heeft ten minste n kleuren nodig, en het kán ook met n: elke knoop een andere kleur. Merk op dat *elke* graaf met n knopen met n kleuren gekleurd kan worden.
c. Genereer alle m^n verschillende kleuringen van de graaf en controleer van elk daarvan of deze voldoet aan de restrictie: loop dus alle buurparen af en controleer of buren een andere kleur hebben. Het genereren van de kleuringen kan bijvoorbeeld stap voor stap gebeuren, zoals bij opgave 5. wordt voorgesteld.

opgave 3.
a. Merk op: als $m \geq n$ is een kleuring met hooguit m kleuren altijd mogelijk (het kan nl. met n): geef elke knoop een andere kleur. Als $m < n$ moet je echt wat doen.
Kleur de knopen een voor een, *bijvoorbeeld* in de volgorde $1, 2, \ldots, n$. Probeer de aan de beurt zijnde knoop te kleuren met achtereenvolgens de kleuren 1 t/m m (*bijvoorbeeld*). Controleer of de buren van deze knoop allemaal een andere kleur hebben dan de kleur die je zojuist probeert. Zo ja, dan kan de knoop deze kleur krijgen en ga je verder op dezelfde manier met de volgende knoop. Zo nee, probeer dan de volgende knoop. Als je alle kleuren voor een bepaalde knoop geprobeerd hebt moet de kleur van de vorige knoop herzien worden.
b. Duidelijk is dat beide grafen met 3 kleuren gekleurd kunnen worden. Op de ene manier vindt je direct een goede kleuring, op de andere manier moet je eerdere keuzes herzien

om tot een oplossing te komen.

opgave 4.
a. Het Latijns vierkant van orde 3 is:

```
1  2  3
2  3  1
3  1  2
```

Doe de rest zelf.
b. Recursief backtracking algoritme:

```
void latijnsvierkant (int n, int A[n][n], int i, int j) {
   int getal, k;
   bool okee;

   if ( i == n )
       drukaf (n, A);
       // i.pv. afdrukken zou je hier ook het aantal
       // Latijnse vierkanten kunnen tellen
   else {
       for ( getal = 1; getal <= n; getal++ ) {
       // controleer of getal al in rij i of kolom j staat
           okee = true;
           for ( k = 0; k < j; k++ ) {
               if ( A[i][k] == getal )
                  okee = false;
           } // for
           if ( okee )
               for ( k = 0; k < i; k++ ) {
                   if ( A[k][j] == getal )
                       okee = false;
               } // for
           if ( okee ) {
               A[i][j] = getal;
               if ( j < n-1 )
                   latijnsvierkant (n, A, i, j+1);
               else
                   latijnsvierkant (n, A, i+1, 1);
           } // if
        } // for
    } // else
} // einde
```

opgave 5.
We gebruiken in main `int part[MAX]` voor het opslaan van de partities, met aantaldelen (het aantal delen waaruit de partitie bestaat) $<$ `MAX` We laten `part`[0] ongebruikt: de partitie begint in `part`[1]. Zie volgende pagina voor de betreffende functie.

```
void partitie (int part[ ], int gedaan, int aantaldelen, int getal,
               int vanaf) {
// Maakt alle partities van getal in aantaldelen, elk deel >= vanaf.
// Al gedaan: gedaan delen. Resultaat komt steeds in part.

  int j;
  if ( ( getal == 0 ) && ( aantaldelen == 0 ) )
     drukaf (part,gedaan); // nieuwe partitie gevonden
  else
     if ( ( getal != 0 ) && ( aantaldelen != 0 ) )
       for ( j = vanaf; j <= getal; j++ ) {
         // of j <= getal - (aantaldelen-1) * vanaf
          part[gedaan+1] = j;    // volgend deel wordt j
          partitie(part, gedaan+1, aantaldelen-1, getal-j, j)
       }  // for
}  // partitie
```

Aanroep: `partitie (part,0,k,n,1);`
Voorbeeld: `partitie (part,0,3,17,5)` maakt alle partities van 17 in 3 delen waarbij elk deel ≥ 5 is, te weten $17 = 5 + 5 + 7 = 5 + 6 + 6$.

opgave 6.

```
void langste(int A[ ], int n, int vanaf, int deelrij[ ],
             int lengte, int & max){
   int i;
   for ( i=vanaf; i<n; i++ ){
       if ( deelrij[lengte] < A[i] ) { // consistent: nog steeds stijgend
           deelrij[lengte+1] = A[i];   // uitbreiden dus
           if ( lengte+1 > max )       // langste deelrij tot nu toe?
               max = lengte+1;
           langste(A, n, i+1, deelrij, lengte+1, max);
       }
   }
}
```

Aanroep: langste(A, n, 0, deelrij, 0, 0);
De eerste aanroep gaat goed omdat `deelrij[0]=0`, de $A[i]$ groter dan 0 zijn en de echte deelrij pas op positie 1 begint.

opgave 7.
Bedenk dat we tijdens het college een deelverzameling als volgt stap voor stap uitbreiden: aan een goede deelverzameling van de objecten 1 t/m i voegen we achtereenvolgens $i+1$ of $i+2$ of $\ldots$ n toe. Zie verder:
`http://liacs.leidenuniv.nl/~vlietrvan1/algoritmiek/voorjaar2016/werkcollege/WG6-2015-kzp`

opgave 8.
Het backtracking algoritme voor het handelsreizigersprobleem werkt in principe hetzelfde als het exhaustive search algoritme dat in hoorcollege 5 is behandeld. Alleen wordt nu tijdens het opbouwen van een route ook gecontroleerd of de lengte van de huidige deelroute niet al hoger (of minstens zo hoog) is als de beste=minimale complete route tot nu toe.
Kortom: we beginnen in knoop 1 (of knoop a): `route[0]`=1, met `huidiggewicht`=0. We breiden de route stap voor stap uit: `route[1]`, `route[2]`, enzovoort. Voor elke positie `pos` proberen we alle mogelijke, nog niet gebruikte knopen voor `route[pos]`. We houden dus bij welke knopen al gebruikt zijn, bijvoorbeeld met een bool-array `gehad`.[1] Daarnaast kunnen we, om dubbele routes te voorkomen, eisen dat knoop 2 vóór knoop 3 wordt bezocht: we mogen alleen knoop 3 aan de route toevoegen als knoop 2 er al in staat. Ten slotte verhogen we meteen `huidiggewicht` met het gewicht van de tak van de vorige knoop in de route naar de zojuist aan de route toegevoegde knoop. Als `huidiggewicht`$\geq$`best` of als een andere controle negatief uitpakt, doen we meteen een stap terug en herzien we onze keuze voor `route[pos]`. Zo niet, dan gaan we recursief verder met `route[pos+1]`.
Wanneer we in de recursie zijn uitgekomen bij `pos`=n, vullen we `route[n]`=1 in, om de Hamiltonkring te sluiten, verhogen `huidiggewicht` met het gewicht van de tak tussen de vorige knoop en knoop 1, en vergelijken `huidiggewicht` met `best`. Wanneer `huidiggewicht`$<$`best`, passen we de waarde van `best` aan.
Doordat we ook de lengte van de huidige deelroute bijhouden, kan dit backtracking algoritme het opbouwen van een route afbreken, als die route geen verbetering meer kan opleveren ten opzichte van de beste, reeds gevonden oplossing. Daardoor kan dit algoritme sneller klaar zijn dan het exhaustive search algoritme, dat de lengte van een route pas berekent als de route compleet is. Dit snoeien van de zoekboom werkt uiteraard alleen goed, wanneer alle gewichten ≥ 0 zijn.
Zie verder:
`http://liacs.leidenuniv.nl/~vlietrvan1/algoritmiek/voorjaar2018/werkcollege/WG6-2018-tsp`

[1] Als de graaf niet compleet zou zijn, zouden we ook nog kunnen controleren of er daadwerkelijk een tak bestaat tussen de vorige knoop en de zojuist aan de route toegevoegde knoop.