ALGORITMIEK: antwoorden werkcollege 5

opgave 1.
a. Brute force algoritme, direct afgeleid uit de observatie: loop v.l.n.r. door de tekst; als je een A tegenkomt op plek i ($0 \leq i < n-1$), loop dan van daaruit naar rechts en tel het aantal B's; deze aantallen voor alle A's sommeren levert het gevraagde aantal substrings. In C++:

```
int teller = 0;
for ( int i=0; i<n-1; i++ ) {
    if ( text[i] == 'A' ) {
        for ( int j=i+1; j<n; j++ ) {
            if ( text[j] == 'B' ) {
                teller++;
            } // if B
        } // for j
    } // if A
} // for i
// teller geeft nu het gevraagde aantal substrings
```

In de worst case (dat komt voor als `text` uit louter A's bestaat), is het aantal karaktervergelijkingen gelijk aan:

$$n + n - 1 + n - 2 + \cdots 3 + 2 = \frac{1}{2} n * (n+1) - 1,$$

dus kwadratisch.
b. We kunnen een slimmer algoritme schrijven, dat elk karakter uit de tekst maar één keer leest. We lopen nog staads van links naar rechts door de tekst. Merk op dat op het moment dat we een B tegenkomen, het aantal sustrings beginnend met een A en deze B als eindkarakter, precies het aantal A's is dat links van B staat, en die je dus al gelezen hebt. We houden derhalve een tellertje bij dat het aantal gelezen A's bijhoudt, en zodra we een B tegenkomen updaten we de totaalteller met het aantal tot dusver gelezen A's. In C++:

```
int totaalteller = 0;
int Ateller = 0;
for ( int i=0; i<n; i++ ) {
    if ( text[i] == 'A' )
        Ateller++;
    if ( text[i] == 'B' )
        totaalteller+=Ateller;
} // for i
// totaalteller geeft nu het gevraagde aantal substrings
```

Dit algoritme doet altijd $2n$ vergelijkingen, en is dus lineair.

opgave 2.
Bereken eerst S, de som van de n getallen. Als S oneven is, dan kun je stoppen, want dan heeft het probleem zeker geen oplossing. Als S even is, genereer dan alle deelverzamelingen (dat zijn er maximaal 2^n) totdat een deelverzameling met som der elementen

$\frac{S}{2}$ wordt gevonden, of totdat alle deelverzamelingen bekeken zijn. In het eerste geval zijn die deelverzameling en de verzameling van de resterende getallen de gevraagde opdeling. In het andere geval is er geen oplossing. Merk op dat je kunt volstaan met het genereren van alleen de deelverzamelingen met niet meer dan $\frac{n}{2}$ elementen.

opgave 3.
a. Laat s de som van de getallen in een rij zijn. In het magisch vierkant is de som der elementen van elke rij dan gelijk aan s (evenals de som der elementen van elke kolom, en van de twee hoofddiagonalen). Als we nu alle getallen van alle rijen (dus alle getallen uit het vierkant) optellen krijgen we:

$$s \times n = 1 + 2 + \cdots n^2 - 1 + n^2 = \frac{1}{2}n^2(n^2 + 1)$$

Hieruit volgt dat $s = \frac{1}{2}n(n^2 + 1)$.
b. Nummer de n^2 posities in een n bij n array van 1 t/m n^2. Genereer een permutatie van de getallen 1 t/m n^2 en zet ze in die volgorde in het array (op de plekken 1 t/m n^2). Controleer dan of aan de eisen voor een magisch vierkant is voldaan door rijsommen, kolomsommen en hoofddiagonaalsommen te bepalen en te kijken of die alle gelijk aan s (zie **a.**) zijn. Zo ja, dan is een magisch vierkant gevonden; zo nee, genereer de volgende permutatie.

opgave 5.
a.

```
bool ALGORITME DFSAcyclisch (G)
// Implementeert DFS wandeling door gegeven graaf,
// en controleert of deze acyclisch is
// Invoer: Ongerichte graaf G = (V,E)
// Uitvoer:
// * Graaf G met zijn knopen genummerd in de volgorde
//   waarin ze bij DFS wandeling voor het eerst worden ontdekt
// * true: als graaf acyclisch is
//   false: als graaf kringen bevat; in dat geval worden de knopen
//     in de gevonden kringen afgedrukt

{ for  elke knoop v in V  do
    mark[v] = 0;   // nog niet bezocht
  od
  teller = 0;
  acyclisch = true;
  for  elke knoop v in V  do
    if  mark[v] == 0  then
      dfs (v, null);  // v is wortel in DFS boom, dus heeft geen ouder
    fi
  od
  return acyclisch;
}
```

```
dfs (v, u)
// Bezoekt recursief alle nog onbezochte knopen die via een pad
// met v zijn verbonden, en nummert deze in de volgorde waarin
// ze worden ontdekt, met globale variabele 'teller'
// Wanneer een kring ontdekt wordt, drukt het de knopen in de kring af.
{ teller ++;
  mark[v] = teller;
  ouder[v] = u;  // ouder in DFS boom
  for  elke buurknoop w van v  do
    if  mark[w] == 0  then
      dfs (w, v);
    else  // back edge, dus een kring gevonden; druk knopen af
      acyclisch = false;
      printpad (w, v);
      print w;
    fi
  od
}

printpad (w, v)
// print recursief het pad van w naar v,
// zoals dat is opgeslagen in array ouder
{ if (w != v)  then
    printpad (w, ouder[v]);
  fi
  print (v);
}
```

b. Het is voldoende om
• in ALGORITME BFSAcyclisch de globale variabele 'acyclisch' te initialiseren en te retourneren, net als in DFSAcyclisch,
• en om in functie 'bfs' de globale variabele 'acyclisch' false te maken als (mark[w] != 0), bij de else van de if in de functie dus.
c. Nee, het is niet zo dat een van de twee algoritmes altijd minstens zo snel een kring ontdekt als de ander.
Bij de ene graaf wordt een kring eerder met DFS ontdekt, bijvoorbeeld als de kring diep in de eerste subboom van de wortel in de DFS boom zit.
Bij een andere graaf wordt een kring eerder met BFS ontdekt, bijvoorbeeld als de kring vlak bij de wortel in de laatste subboom van de wortel in de BFS boom zit.

opgave 6.

```
bool ALGORITME DFSSamenhangend (G)
// Implementeert DFS wandeling door gegeven graaf,
// en controleert of deze samenhangend is
// Invoer: Ongerichte graaf G = (V,E)
// Uitvoer:
// * Graaf G met zijn knopen genummerd in de volgorde
//   waarin ze bij DFS wandeling voor het eerst worden ontdekt
```

```
// * true: als graaf samenhangend is
//   false: als graaf niet samenhangend is

{ for  elke knoop v in V  do
    mark[v] = 0;   // nog niet bezocht
  od
  teller = 0;
  component = 0;
  for  elke knoop v in V  do
    if  mark[v] == 0  then
      component ++;
      dfs (v);
    fi
  od
  if  component == 1  then
    return true;
  else
    return false;
  fi
}
```

```
dfs (v)
// Bezoekt recursief alle nog onbezochte knopen die via een pad
// met v zijn verbonden, en nummert deze in de volgorde waarin
// ze worden ontdekt, met globale variabele 'teller'
{ teller ++;
  mark[v] = teller;
  compon[v] = component;
  for  elke buurknoop w van v  do
    if  mark[w] == 0  then
      dfs (w);
    fi
  od
}
```

opgave 7.

a. Er zijn n knopen; elke knoop kan met m mogelijke kleuren gekeurd 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 8.
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 (*). Zo ja, dan heeft het geen zin op verder te gaan, dus probeer op plek (i, j) de volgende waarde. Zo nee, breid de deeloplossing dan 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 nog gevuld moet worden, een afschatting maken van hetgeen ten minste de rijsom/kolomsom zal worden. Overigens kun je ook op soortgelijke manier een afschatting geven van de maximaal te bereiken rijsom/kolomsom, 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 9.
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 10.
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 11.

```
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);
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.