1. (a) De buitenste for-lus kent $N-1=5$ iteraties. Na iedere iteratie ziet de rij getallen er als volgt uit:

i	rij na i^{e} iteratie
1	3 2 5 1 4 6
2	2 3 1 4 5 6
3	2 1 3 4 5 6
4	1 2 3 4 5 6
5	1 2 3 4 5 6 (hetzelfde dus)

(b) Na de eerste iteratie van de buitenste for-lus is het grootste getal naar achteren geborreld. Na de tweede iteratie van de buitenste for-lus zijn (in ieder geval) de grootste twee getallen naar achteren geborreld. In het algemeen zijn na de i^{e} iteratie van de buitenste for-lus de grootste i getallen naar achter geborreld, en ze staan daar in de juiste volgorde. Dit zijn dus de getallen op posities $N-i+1,\dots,N$ (inderdaad, dit zijn i posities). De getallen op posities $1,\dots,N-i$ zijn allemaal kleiner dan of gelijk aan A[N-i+1].

De derde bewering is dus een invariant voor de buitenste for-lus op positie (*).

(c) In de buitenste for-lus loopt de variabele i van 1 tot en met $N-1$. In de laatste iteratie is dus $i=N-1$. Aan het eind van die iteratie is de invariant geldig. Als we in de invariant de waarde $i=N-1$ invullen, krijgen we:

$A[N-(N-1)+1\dots N]=A[2\dots N]$ is van klein naar groot gesorteerd en
$A[1\dots N-(N-i)]=A[1\dots 1]=A[1]$ is hoogstens zo groot als $A[N-(N-1)+1]=A[2]$.

Dit betekent dat $A[1\dots N]$, ofwel het hele array A, van klein naar groot gesorteerd is.

(d) In de i^{e} iteratie van de buitenste for-lus loopt de variabele j in de binnenste for-lus van 1 tot en met $N-i$. Dit zijn dus $N-i$ iteraties.

(e) In de eerste iteratie van de buitenste for-lus is $i=1$ en kent de binnenste for-lus volgens het vorige onderdeel $N-1$ iteraties.

In de tweede iteratie van de buitenste for-lus is $i=2$ en kent de binnenste for-lus $N-2$ iteraties.

In de derde iteratie van de buitenste for-lus is $i=3$ en kent de binnenste for-lus $N-3$ iteraties.

Enzovoort, tot en met de $(N-1)^{\mathrm{e}}$ = laatste iteratie van de buitenste for-lus, waarin $i=N-1$ en de binnenste for-lus nog $N-i=N-(N-1)=1$ iteratie kent.

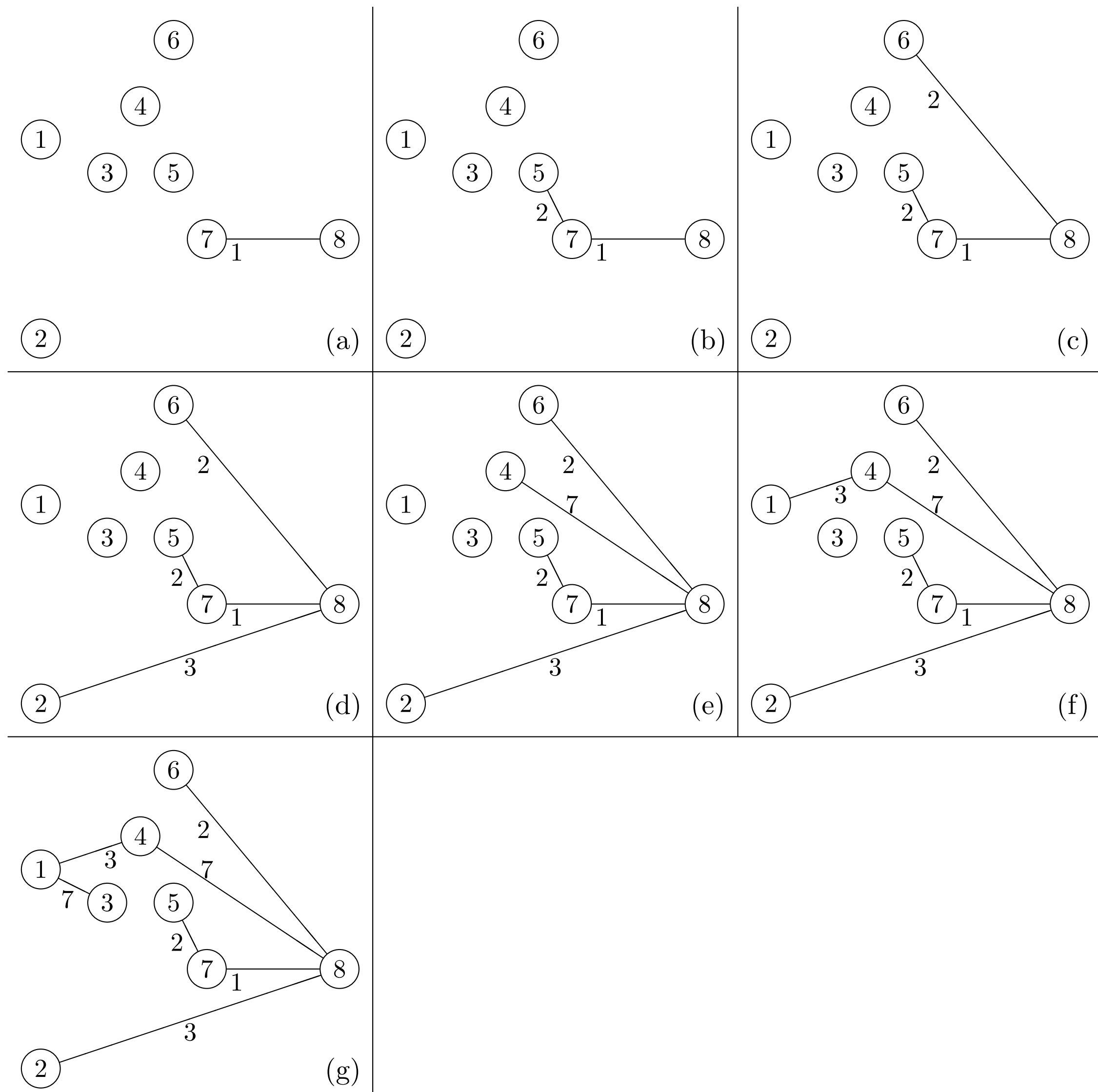
Het totaal aantal iteraties van de binnenste for-lus is dus $(N-1)+(N-2)+(N-3)+\cdots+1$. Dit zijn $N-1$ termen met als gemiddelde waarde $\frac{(N-1)+1}{2}=\frac{N}{2}$. De som is dus gelijk aan $(N-1)\times\frac{N}{2}$ iteraties.

Voor iedere iteratie van de binnenste for-lus is (hoogstens) een constante hoeveelheid tijd nodig. Verder gebeurt er niets spannends in het algoritme. De tijdscomplexiteit wordt dus volledig bepaald door het aantal iteraties van de binnenste for-lus. Dit is $(N-1)\times\frac{N}{2}=\frac{1}{2}N^2-\frac{1}{2}N$ De achterste term $-\frac{1}{2}N$ valt voor grote N weg tegen de voorste term $\frac{1}{2}N^2$. De constante $\frac{1}{2}$ van de voorste term is niet van belang voor de complexiteit. De complexiteit van het algoritme is dus $\mathcal{O}(N^2)$.

(f) Nee, want voor iedere rij getallen doorloopt de buitenste for-lus $N-1$ iteraties. En in de i^{e} iteratie van de buitenste for-lus doorloopt de binnenste for-lus altijd $N-i$ iteraties. Dit is onafhankelijk van de waardes die we tegenkomen.

P.S.: je kunt het algoritme wel slimmer maken door uit de buitenste for-lus te springen als het array al helemaal gesorteerd blijkt te zijn, maar dat gebeurt dus niet standaard.

2. (a) We passen het algoritme uit het boek (het algoritme van Prim) toe. We beginnen met de tak met het kleinste gewicht. Iedere volgende stap breiden we de huidige boom uit met een aangrenzende tak, die geen kring vormt met de eerder gekozen takken, en die een zo klein mogelijk gewicht heeft. Hierbij krijgen we achtereenvolgens de volgende bomen:



Het eindresultaat is een minimale opspannende boom. In de tweede stap konden we kiezen uit twee takken met hetzelfde (kleinste) gewicht 2. In dat geval hebben we willekeurig de tak 5–7 gekozen. In de volgende stap kozen we alsnog de tak 6–8.

(b) Ja, zo'n Hamilton pad bestaat, bijvoorbeeld het pad 4–1–3–2–5–7–8–6–9–10.

(c) Een kortste route voor de handelsreiziger is 1–3–2–7–5–4–8–10–9–6–1. Deze route heeft lengte 57.

(d) Een niet-deterministisch polynomiaal algoritme voor de beslissingsvariant van het handelsreizigersprobleem is:

i. Genereer een willekeurige permutatie van de getallen $1, \ldots, N$.

ii. Bereken het totale gewicht van de route die de knopen afloopt in de volgorde van de permutatie, en aan het eind weer terugkeert van de laatste knoop naar de eerste knoop.

(Als een of meer van de takken op deze route niet in de graaf voorkomen, wordt het totale gewicht 'oneindig'.)
Als het berekende totale gewicht hoogstens K is, antwoord dan "ja". Antwoord anders "nee".

3. Makkelijkst is 'zoeken van een element in een gesorteerd array'. Dat kan in $\mathcal{O}(\log N)$ tijd (met binair zoeken).

   Vervolgens komt 'het bepalen van een minimale opspannende boom in een ongerichte graaf'. Dit kan in polynomiale tijd, en die tijd is minstens lineair, dus minstens $\mathcal{O}(N)$.

   Vervolgens komt het handelsreizigersprobleem. Dit probleem is NP-volledig. Dat betekent dat het wel op te lossen is, maar (voor zover we nu weten) niet in polynomiale tijd.

   Ten slotte komt het correspondentieprobleem. Dat probleem is niet oplosbaar.

4. Stel dat het stopprobleem wel oplosbaar is. Ofwel, stel dat er een programma Q bestaat dat, gegeven een programma R en een invoer X, binnen eindige tijd kan bepalen of programma R zal stoppen voor invoer X.

   Dan bouwen we met behulp van Q een nieuw programma S, als volgt.

   > S verwacht als invoer een computerprogramma W, en maakt hier een kopie van. De oorspronkelijke invoer noemen we R en de kopie noemen we X.
   >
   > Vervolgens gebruikt S de variabelen R en X (die dus allebei gelijk zijn aan W) als invoer voor het programma Q (waarvan we dus veronderstellen dat het bestaat). We laten nu Q draaien op deze invoer, en Q beantwoordt de vraag of computerprogramma R stopt, wanneer het X als invoer krijgt. Ofwel, Q beantwoordt de vraag of computerprogramma W stopt, wanneer het zichzelf (zijn eigen code) als invoer krijgt.
   >
   > De uitvoer hiervan wordt gebruikt in de volgende stap van het programma S dat we aan het bouwen zijn. Als Q de uitvoer "ja" geeft (wat betekent dat W stopt met zichzelf als invoer), dan gaat S in een oneindige lus. Als Q de uitvoer "nee" geeft (wat betekent dat W niet stopt met zichzelf als invoer), dan stopt S.

   Tot zover de beschrijving van het nieuwe programma S. (Zie Figuur 8.7 op blz. 203 van het boek.) Ervanuitgaande dat Q bestaat, is S een keurig programma.

   We voeren nu de code van S als invoer aan zichzelf. Dit is legaal, want S verwacht een computer programma als invoer. Wat gebeurt er dan?

   Er wordt een kopie van S gemaakt, en Q beantwoordt de vraag of S stopt met zichzelf als invoer. Het antwoord van Q is "ja" of "nee".

   Als het antwoord "ja" is, gaat S in een oneindige lus, en stopt dus niet. Dat is in tegenspraak met het antwoord van Q.

   Als het antwoord "nee" is, stopt S netjes. Ook dat is in tegenspraak met het antwoord van Q.

   In beide gevallen stuiten we op een tegenspraak. Ergens moet dus iets mis zijn met S. Al onze stappen om S vanuit Q te construeren waren echter legaal: kopiëren van de invoer, draaien van Q, eventueel in een oneindige lus geraken. Er moet dus iets mis zijn met Q. De aanname dat Q bestaat, klopte kennelijk niet. Er bestaat kennelijk geen algoritme dat voor een willekeurig programma R en invoer X bepaalt of R stopt voor X. Het stopprobleem is dus niet oplosbaar.

5. (a) We leggen het woord W eerst onder de eerste M letters van de tekst T. Dan kijken we letter voor letter (van links naar rechts) of W gelijk is aan dat deel van de tekst.

Als dat inderdaad het geval is, kunnen we stoppen. Als daarentegen een bepaalde letter van W niet gelijk is aan de corresponderende letter van T, hoeven we de volgende letters van W niet meer te vergelijken. We schuiven W één positie door naar rechts, zodat het onder de 2e tot en met de $(M+1)^{e}$ letter van T komt te liggen. We gaan dan opnieuw letter voor letter (van links naar rechts) kijken of W gelijk is aan dat deel van de tekst.

Zo gaan we door, totdat we uiteindelijk W inderdaad ergens in de tekst vinden, of totdat we W voorbij de laatste letter van de tekst schuiven.

In pseudo-code:

```
Tbeginpos = 1;
Gevonden = false;
while (!Gevonden and Tbeginpos<=N-M+1)
do   Wpos = 1;
     Tpos = Tbeginpos;
     NogOK = true;
     while (NogOK and Wpos<=M)
     do   if (W[Wpos]==T[Tpos])
          then Wpos ++;
               Tpos ++;
          else NogOK = false;
          fi
     od

     if (NogOK)
     then Gevonden = true;
     else Tbeginpos ++;
     fi
od

if (Gevonden)
then write "woord W komt voor";
else write "woord W komt niet voor";
fi
```

(b) In het slechtste geval komen alle letters van W steeds overeen met de tekstletters, tót de laatste letter, waar het misgaat.

We zullen dan posities $1, 2, \ldots, (N-M+1)$ van de tekst T als beginpositie van het woord W proberen. Gewoon omdat we W alsmaar niet tegenkomen. De buitenste while-lus doorloopt dan $N-M+1$ iteraties.

Verder doorloopt de binnenste while-lus voor iedere beginpositie (ofwel, voor iedere iteratie van de buitenste while-lus) M iteraties: een iteratie voor iedere letter van W. De eerste $M-1$ letters van W blijken immers steeds gelijk te zijn aan de corresponderende tekstletters.

In totaal doorloopt de binnenste while-lus in het slechtste geval dus $(N-M+1) \times M = N \times M - M \times M + M$ iteraties. Omdat we ervan uit mogen gaan dat M veel kleiner is dan N, is dit $\mathcal{O}(N \times M)$.

Het meeste werk vindt plaats in de binnenste while-lus. De hoeveelheid werk per iteratie van de binnenste while-lus is (hoogstens) constant (één vergelijking en twee of één toekenningen). Het totaal aantal iteraties van de binnenste while-lus bepaalt dus de hoeveelheid werk. Zoals we gezien hebben is dit $\mathcal{O}(N \times M)$. Dit is dan ook de tijdscomplexiteit van het algoritme.

(c) Een voorbeeld van een slechtste geval dat overeenkomt met de redenering van het vorige onderdeel, is $T =$ AAAAA AAAAA AAAAA (de spaties in T staan hier louter voor de leesbaarheid; ze zitten dus niet in T zelf) en $W =$ AAAAB.

(d) Stel dat we voor een blok G de vingerafdruk willen berekenen, en dat we geen gebruik kunnen maken van de vingerafdruk van het vorige blok. Dan moeten we waarschijnlijk alle M cijfers van G aflopen. Dat kost $\mathcal{O}(M)$ tijd. Als we pech hebben, moeten we dat voor alle $N - M + 1$ blokken doen, zodat de tijdscomplexiteit van dit algoritme $\mathcal{O}(N \times M)$ wordt, net als voor het naïeve algoritme. We zouden dus geen tijd besparen.

(e)

$$\begin{aligned} &\text{VingerAfdruk}(30787) = 30787 \bmod 101 = (30780 + 7) \bmod 101 \\ &= (10 \times 3078 + 7) \bmod 101 = (10 \times (63078 - 60000) + 7) \bmod 101 \\ &= (10 \times (63078 \bmod 101) - (600000 \bmod 101) + 7) \bmod 101 \\ &= (10 \times \text{VingerAfdruk}(63078) - (600000 \bmod 101) + 7) \bmod 101 \end{aligned}$$

We moeten dus de vingerafdruk van het vorige blok met 10 vermenigvuldigen, daar 600000 mod 101 vanaf trekken en 7 bij optellen. En vervolgens het resultaat modulo 101 doen. Dit zijn allemaal eenvoudige operaties, die we in constante tijd kunnen uitvoeren.

Het is hierbij van belang dat we de waarde 600000 mod 101 niet rechtstreeks hoeven uit te rekenen (wat 'veel' werk zou zijn), maar uit een simpele tabel kunnen halen. Op deze plek in de berekening hebben we namelijk bij ieder blok óf 0 mod 101, óf 100000 mod 101, óf 200000 mod 101, ..., óf 900000 mod 101 nodig. Dit zijn 10 waardes die we eenvoudig van tevoren kunnen uitrekenen en in een tabel stoppen.

(f) Volgens de definitie is de vingerafdruk van een blok G gelijk aan G mod K. Dit is een waarde uit de het rijtje 0,1,2,...,K-1. Er zijn dus K verschillende vingerafdrukken. De kans dat een willekeurig blok van lengte M dezelfde vingerafdruk heeft als het woord dat we zoeken is dus 1 op K. In ons voorbeeld is dat $\frac{1}{101}$, wat ongeveer 1% is.

(g) Omdat het dan niet mogelijk is om expres een instantie (een combinatie van T en W) te construeren, waarvoor het algoritme altijd slecht zal werken. Je mag er dus vanuit gaan dat gemiddeld maar 1 op de K blokken in de tekst dezelfde vingerafdruk heeft als W. Alleen voor die blokken moet je controleren of ze echt gelijk zijn aan W, en dat is niet veel werk.

Als K niet random gekozen zou worden, zou je bij een woord W een tekst T kunnen construeren die veel blokken bevat met dezelfde vingerafdruk als W. Bij al die blokken zou het algoritme vervolgens moeten controleren of ze echt gelijk zijn aan W, en dat zou veel werk worden.