10:05 **Uitwerking tentamen Algoritmiek, dinsdag 11 juni 2013**

1(a)

Toestanden:

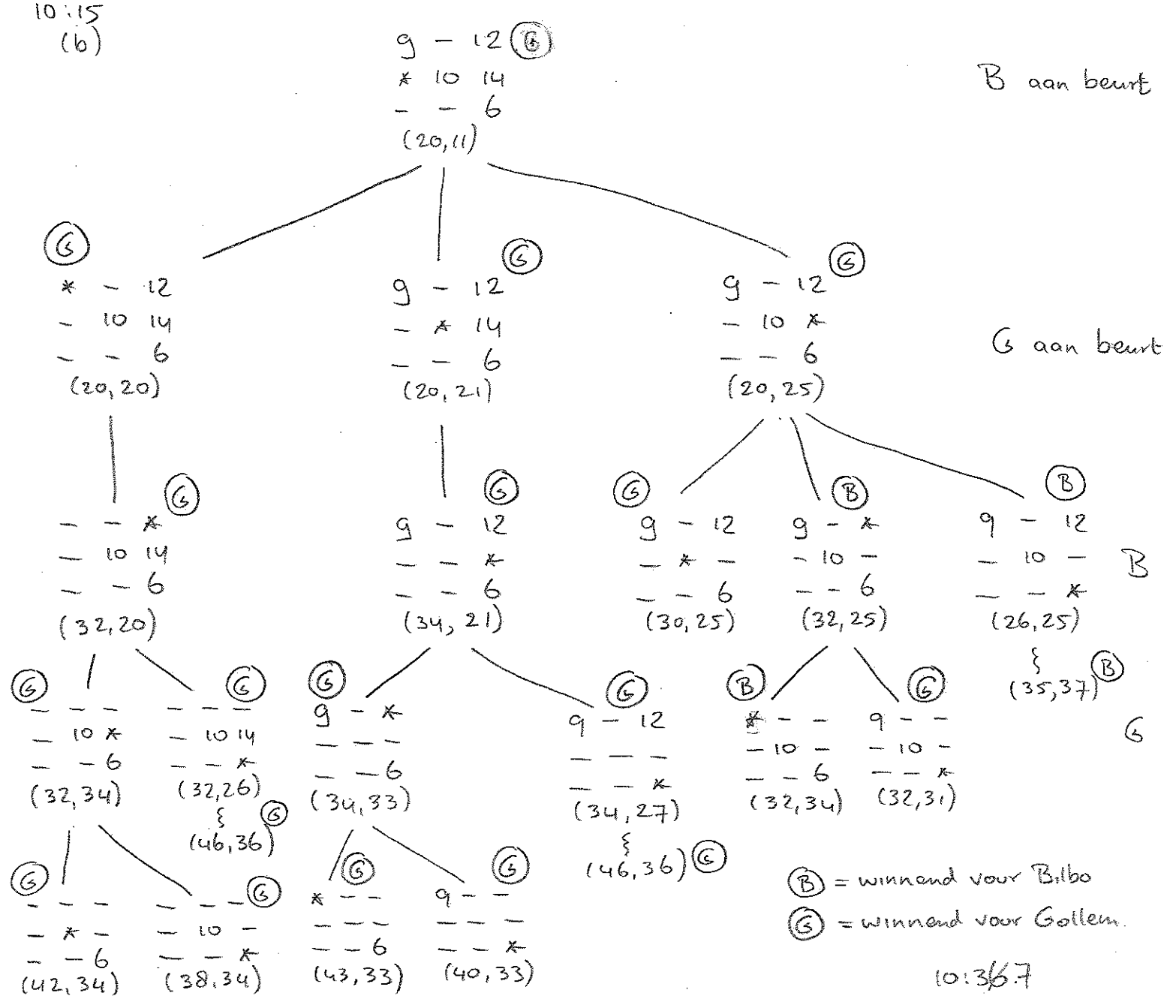
* m * n tabellen met
  - getallen
  - lege vakjes
  - één speciaal (leeg) vakje
* score van beide spelers
* speler aan de beurt
* eventueel: speler die begonnen is (voor bij gelijkspel), maar dat kun je ook afleiden uit het aantal lege vakjes

Acties

* het kiezen van een getal in de rij of kolom van het speciale vakje.
  Resultaat is
  - dat getal opgeteld wordt bij score van speler aan de beurt
  - dat vakje van dit getal leeg en speciaal wordt
  - dat speler aan de beurt wisselt.

10:15

(b)



10:36.7

(c) 10:37 | Uitwerking tentamen Algoritmiek, dinsdag 11 juni 2013 |

*	2	1
3	100	4
	(0,0)	

Volgens de gretige strategie kiest Gollem hier getal 3, waarna Bilbo getal 100 kan kiezen en zal winnen.
Als Gollem echter aan het begin getal 1 had gekozen, had Bilbo in de volgende zet getal 2 of 4 moeten kiezen, waarna Gollem getal 100 had kunnen kiezen en had gewonnen

10:42

10:44

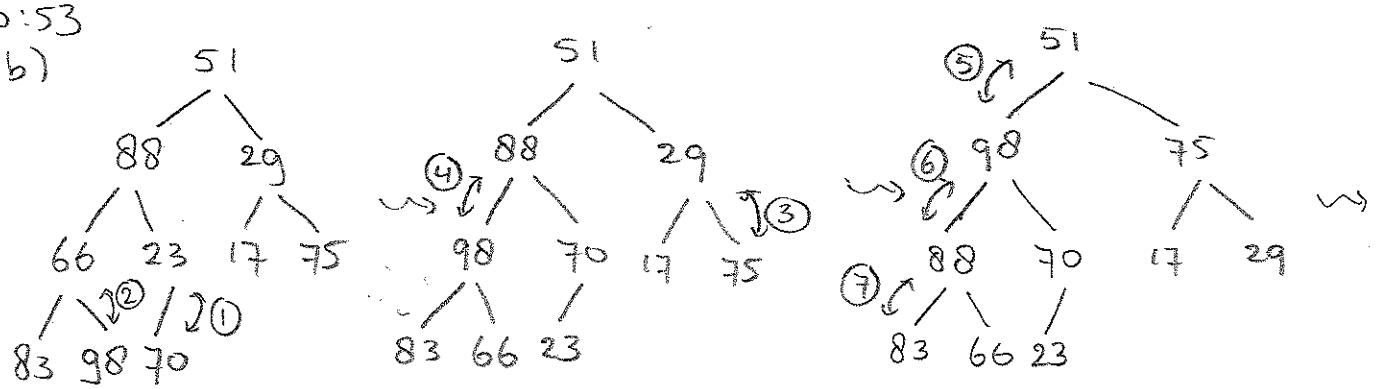
2 (a)

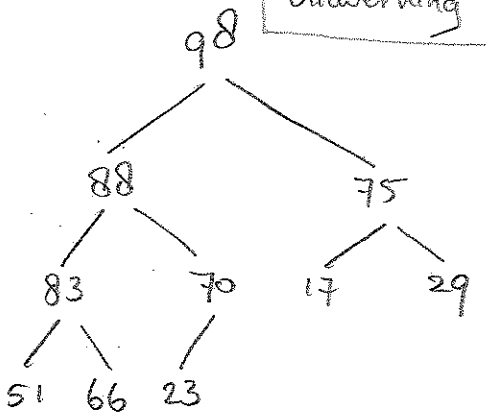
```
bool heap (knoop *wortel)
 // pre: wortel ≠ NULL
{ bool OK;

  OK = true;
  if (wortel → links != NULL)
  { OK = heap (wortel → links);
    if (wortel → waarde < wortel → links → waarde)
      OK = false
  }
  if (wortel → rechts != NULL)
  { if (!(heap (wortel → rechts)))
      OK = false;
    if (wortel → waarde < wortel → rechts → waarde)
      OK = false;
  }
  return OK;
}
```

10:51

10:53

(b)





De hiermee corresponderende rij is

98 88 75 83 70 17 29 51 66 23

11:01

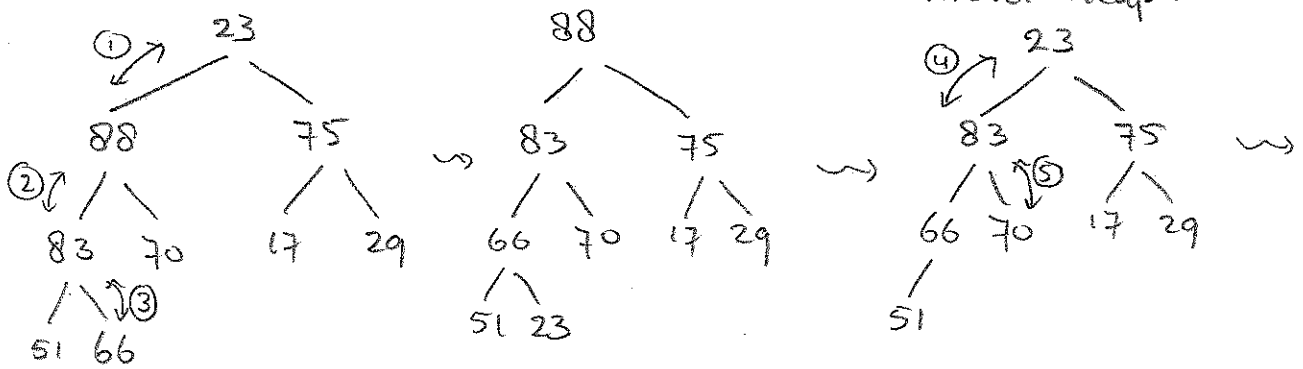
(c)

Als we een rij getallen willen sorteren met HeapSort, stoppen we de rij eerst in een complete binaire boom, en maken daar (b.v. met heapify) een heap van.
Vervolgens verwijderen we herhaaldelijk het maximum uit de heap, totdat de heap leeg is. Uiteraard wordt de boom na iedere verwijdering weer een heap gemaakt. De getallen komen op deze manier in aflopende volgorde uit de heap. Desgewenst kun je die volgorde nog omkeren, om de getallen in oplopende volgorde te krijgen.

11:06

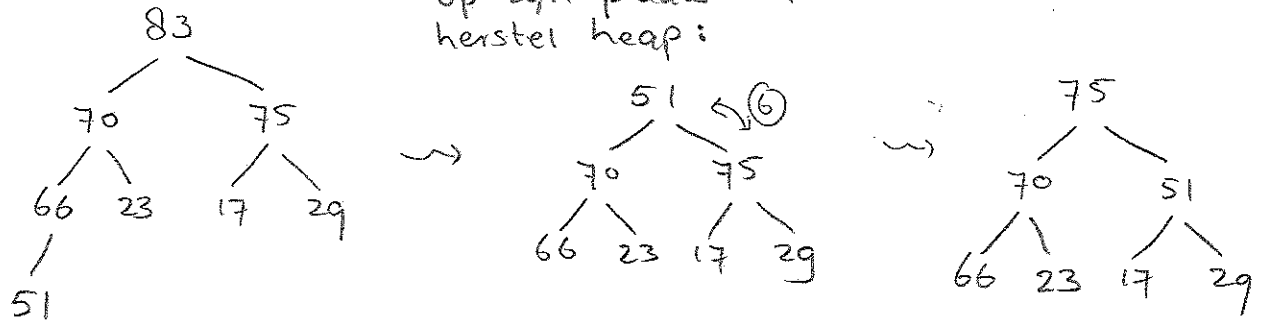
Je kunt ook iedere keer het verwijderde getal achteraan de rij met resterende getallen zetten

Verwijder max (98), zet laatste getal 23 op zijn plaats en herstel heap:



Verwijder max (88), zet laatste getal 23 op zijn plaats en herstel heap:

Verwijder max (83) zet laatste getal 51 op zijn plaats en herstel heap:



Uitwerking tentamen Algoritmiek, dinsdag 11 juni 2013

Hierna ziet de (complete) rij er als volgt uit:

75 70 51 66 23 17 29 | 83 88 98

11:17.

11:18

3 (a)

```
j = n-1;
while (j>=0 && A[j] >= j)
  j--;
return j;
```

11:21

(b)

```
int grootste2 (int A[], int links, int rechts)
// bepaalt grootste index j met links ≤ j ≤ rechts
// waarvoor A[j] < j;
// als zo'n index j niet bestaat: -1.
{ int l1, r1,                    // werkt trouwens ook als
      l2, r2;                    // rechts-links+1 geen 2-macht is.
    j;
  if (links == rechts)
  { if (A[links] < links)
      return links
    else
      return -1;
  }
  else
  { l1 = links;      r1 = (links + rechts) / 2;
    l2 = r1 + 1;     r2 = rechts;
    j = grootste2 (A, l2, r2);
    if (j != -1)
      return j;
    else
    { j = grootste2 (A, l1, r1);
      return j;
    }
  }
} // grootste2
```

11:30

(c)

We kunnen op dezelfde manier als bij (b) de indices l1, r1, l2, r2 berekenen.

Als A[r1] ≥ r1, dan weten we zeker dat A[j] > j voor elke j > r1. Immers, de getallen in A lopen op en zijn verschillend ⇒ ze lopen minstens zo hard op als de indices zelf ⇒ het verschil tussen A[j] en j zal alleen maar toenemen.

In formules: als j > r1, geldt:

$$A[j] - j \geq A[r1] + (j - r1) - j = A[r1] - r1 \geq 0.$$

Ofwel: A[j] ≥ j

In dit geval hoeven we de rechter helft dus niet meer te bekijken.

Als A[r1] < r1, dan is in ieder geval r1 een waarde van j waarvoor A[j] < j. De j's kleiner dan r1 zullen dan nooit de grootste j's met A[j] < j kunnen zijn. In dit geval hoeven we de linker helft dus niet te bekijken.

11:43

Dit resulteert in:

```
int grootste3 (int A[], int links, int rechts)
// bepaalt grootste index j met links ≤ j ≤ rechts waarvoor A[j] < j;
// als zo'n index j niet bestaat: -1
// pre: getallen in A zijn verschillend en oplopend gesorteerd
// werkt ook als rechts - links + 1 geen 2-macht is
{ int l1, r1, l2, r2, j;
  if (links == rechts)
  { if (A[links] < links)
      return links;
    else
      return -1;
  }
  else
  { l1 = links;        r1 = (links+rechts)/2;
    l2 = r1+1;         r2 = rechts;
    if (A[r1] ≥ r1)
      return grootste3 (A, l1, r1);   // grootste3 (A, l1, r1-1) zou
                                      // in principe ook kunnen, maar
                                      // dan moeten we erop letten
                                      // dat l1 > r1-1 kan worden.
    else  // A[r1] < r1
    { j = grootste3 (A, l2, r2);
      if (j != -1)
        return j;
      else
        return r1;
    }
  }
}
```

11:51

Alternatief: kijk naar l2 ipv r1

```
int grootste3 (int A[ ], int links, int rechts)
{ int l1, r1, l2, r2, j;
  if (links == rechts)
  { if (A[links] < links)
      return links;
    else
      return -1;
  }
  else
  { l1 = links;        r1 = (links + rechts)/2;
    l2 = r1+1;         r2 = rechts;
    if (A[l2] >= l2)
      return grootste3 (A, l1, r1);
    else
      return grootste3 (A, l2, r2);
  }
} // grootste3
```

11:58

12:01

Bonus:

Het algoritme van (a)
* kost in het beste geval $O(1)$ tijd, als $A[n-1] < n-1$ (je springt meteen uit de while-lus)
* kost in het slechtste geval $O(n)$ tijd, als elke $A[j] \geq j \Rightarrow$ de while-lus doorloopt alle j's.

Het algoritme van (b)
* kost in het beste geval $O(\lg n)$ tijd, als $A[n-1] < n-1$.
  We roepen de functie grootste2 dan namelijk steeds alleen voor de rechter helft aan, en wel $^2\lg n + 1$ keer.
* kost in het slechtste geval $O(n)$ tijd, als elke $A[j] \geq j$.
  We roepen de functie grootste2 dan namelijk steeds zowel voor de rechter helft als voor de linker helft aan, in totaal $2n-1$ keer.

Het algoritme van (c)
* kost altijd $O(\lg n)$ tijd.
  We roepen de functie grootste3 namelijk altijd voor maar één helft recursief aan, en wel $^2\lg n + 1$ keer.

Dus
* het algoritme van (c) is altijd minstens zo efficiënt als het algoritme van (b)
* als $A[n-1] < n-1$, is het algoritme van (a) het efficiëntst, en zijn de algoritmes van (b) en (c) even (zeer) efficiënt.

* als $A[j] \geq j$ voor elke j, is het algoritme van (c) het efficiëntst, en zijn de algoritmes van (a) en (b) grote O-technisch even (weinig) efficiënt.

12:16.

4 (a)

```
int maxSom (int i, int j, int n)
// bepaal maximale som op wandeling van positie (i,j) naar rij n.
{
    if (i == n)
      return D[i][j];
    else // i<n
      return D[i][j] + max { maxSom (i+1, j, n); maxSom (i+1, j+1, n) }
}
```

Aan te roepen met 'maxSom (1,1,n)'.

Dit algoritme controleert 2^{n-1} complete routes.
Immers: voor (1,1) bestaat het twee mogelijkheden
voor (2,1) en (2,2) bestaat het twee mogelijkheden
enz:
bij de overgang van elke rij naar de volgende rij
(of wel bij uitbreiding van een route met één stap),
wordt het aantal deelroutes twee keer zo groot.
We beginnen met één deelroute: alleen (1,1)
Op rij n hebben we dan 2^{n-1} (complete) routes opgebouwd.

12:26

(b) Voor de Fibonacci getallen geldt dat

Fib (0) = 0
Fib (1) = 1
Fib (n) = Fib (n-1) + Fib (n-2) als $n \geq 2$.

Als je deze recurrente betrekking rechtstreeks zou vertalen in een recursieve formule, zou je veel waardes meerdere keren berekenen

Bij dynamisch programmeren sla je berekende waardes op in een array F, zodat je ze niet nog een keer hoeft te berekenen.

Bij bottom up dynamisch programmeren vullen we eerst F[0] en F[1] in, en vervolgens F[2], F[3], — F[n], als we het n-de Fibonacci getal willen weten.

Bij top down dynamisch programmeren werken we (gewoon) met een recursieve functie, maar slaan we (dus) de berekende waardes op.

Voordeel van dynamisch programmeren boven gewoon recursie bij de berekening van de Fibonacci getallen is dat je veel tijd bespaart, omdat je geen waardes meerdere keren berekent.

12:37 Uitwerking tentamen Algoritmiek, dinsdag 11 juni 2013

(c) als $i=n$ (en $1 \leq j \leq i$) valt er niet veel te lopen:
je bent al op rij n en kunt alleen getal $D[i][j]$ verdienen
als $i<n$ (en $1 \leq j \leq i < n$), heb je vanuit positie (i,j) twee mogelijkheden: naar positie $(i+1,j)$, waar je $S[i+1][j]$ kunt verdienen
óf naar positie $(i+1,j+1)$, waar je $S[i+1][j+1]$ kunt verdienen.
Je wilt een zo hoog mogelijke score halen, dus neem je het maximum van de twee mogelijkheden. In beide gevallen kun je het getal $D[i][j]$ er nog bij optellen

12:44

(d) We willen het array S vanaf rij n 'omhoog' tot rij 1

```
for (j=1; j≤n; j++)
  S[n][j]= D[n][j].
for (i= n-1; i≥1; i--)
   for (j=1; j≤i; j++)
     S[i][j] = D[i][j] + max{ S[i+1][j], S[i+1][j+1] }
```

12:48
12:51

S
22
18 20
9 13 16
1 6 9 6

12:53

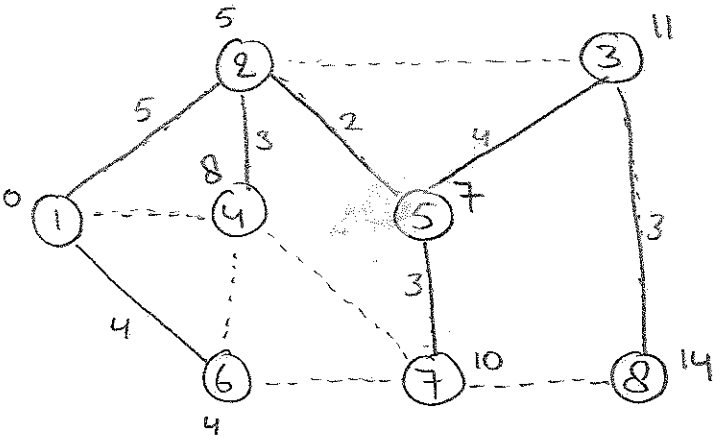
(e)
Iedere keer dat je in het algoritme van (d) het maximum van $S[i+1][j]$ en $S[i+1][j+1]$ bepaalt (voor de berekening van $S[i][j]$), sla je bij positie (i,j) ook de 'kolom' van S op waarbij dat maximum wordt bereikt: j of $j+1$.

Vervolgens kun je vanaf positie $(1,1)$ de optimale route lopen door steeds te kijken naar welke kolom in de volgende rij je moet. (j of $j+1$ dus steeds).

12:59 Alternatief: rechtstreeks vanuit $S[i][j]$, in pseudo-code:

```
i=1; j=1; Print (i,j);
while (i<n)
{ if (S[i][j+1] > S[i][j])
     j++;
  i++; Print (i,j);
}
```

5)



Tabel met kandidaatafstanden na achtereenvolgende iteraties van het algoritme van Dijkstra

1	2	3	4	5	6	7	8	
0	∞	∞	∞	∞	∞	∞	∞	begin met knoop 1
–	5	∞	9	∞	4	∞	∞	kies knoop 6 vanaf knoop 1
–	5	∞	9	∞	–	12	∞	kies knoop 2 vanaf knoop 1
–	–	12	8	7	–	12	∞	kies knoop 5 vanaf knoop 2
–	–	11	8	–	–	10	∞	kies knoop 4 vanaf knoop 2
–	–	11	–	–	–	10	∞	kies knoop 7 vanaf knoop 5
–	–	11	–	–	–	–	16	kies knoop 3 vanaf knoop 5
–	–	–	–	–	–	–	14	kies knoop 8 vanaf knoop 3

klaar.

13:13

De doorgetrokken lijnen in het plaatje hierboven vormen de boom van kortste paden.

13:15

Controle

13:19.