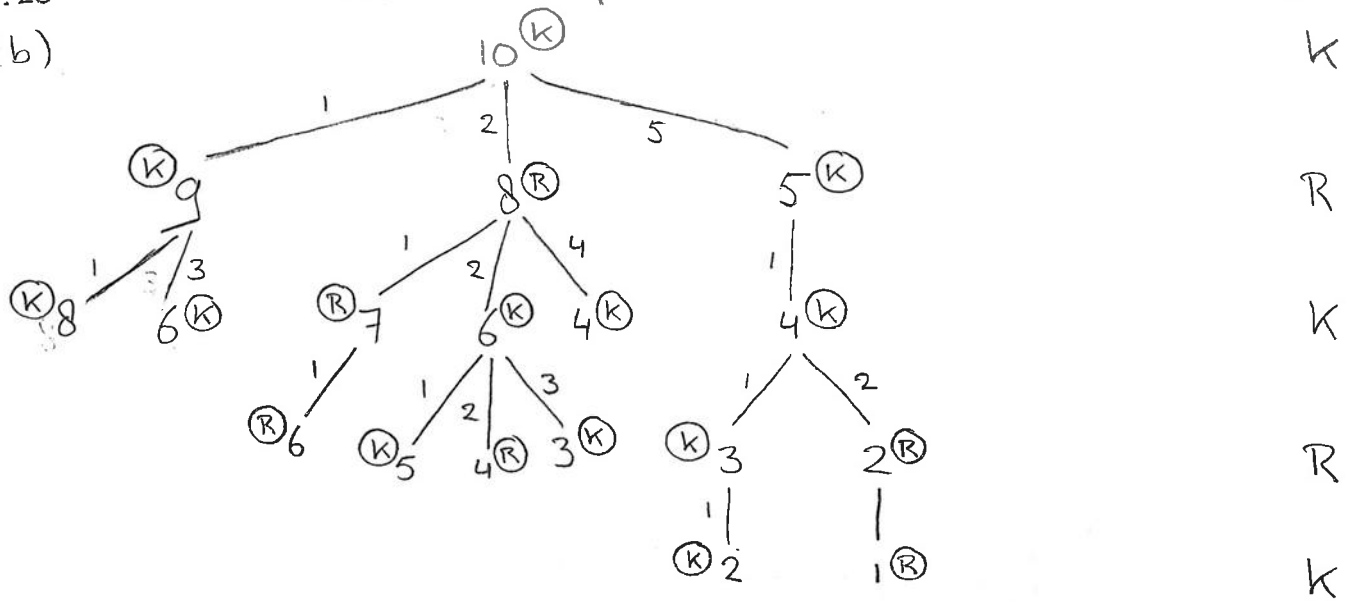
12.20

1(a) Bij dit spel bestaat een toestand uit
- het aantal lucifers dat nog aanwezig is: $m \geq 1$
- de speler die aan de beurt is

Een actie bestaat uit
- het weghalen van een aantal lucifers d, met d een deler van m en $d < m$.

12.26 - het wisselen van de speler aan de beurt.

(b)



12.39

De omcirkelde letter (K) of (R) geeft aan voor welke speler (Kiki of Raemon) een toestand winnend is.

Toestanden met hetzelfde aantal lucifers worden hierboven als hetzelfde beschouwd, en worden daarom maar één keer uitgewerkt, ook als de andere speler aan de beurt is.

In dat laatste geval is de toestand ook winnend voor de andere speler dan die waarvoor de eerdere toestand winnend is.

Kiki wint dus door iedere keer dat zij aan de beurt is een zet te kiezen waarmee ze in een voor haar winnende vervolgtoestand komt. Daar zijn (soms) verschillende mogelijkheden voor. In de eerste zet kan ze bijvoorbeeld 1 lucifer of 5 lucifers weghalen en in beide gevallen winnen

12.48

(c) Als $n=1$ (is oneven) is het spel verliezend voor de speler die begint. Ze kan nog niet eens een eerste zet doen.
Als $n=2$ (is even) is het spel winnend voor de speler die begint. Ze kan als enige mogelijke zet 1 lucifer weghalen, zodat de tegenstander met 1 lucifer blijft zitten en verliest.

De bewering uit de opgave klopt dus voor $n=1$ en voor $n=2$.

Laat $n_0 \geq 2$, en laat gelden voor elke n met $1 \leq n \leq n_0$ dat
* het spel winnend is voor de speler die begint als n even is
* het spel verliezend is voor de speler die begint als n oneven is

(inductiehypothese).
We kijken nu naar het spel met $n = n_0 + 1$

Als $n = n_0 + 1$ is oneven, dan zijn alle delers van n ook oneven. De speler die begint moet dus een oneven aantal $d \geq 1$ lucifers weghalen, zodat er $n - d \leq n_0$ lucifers overblijven, en $n - d$ (oneven − oneven) is even. Volgens de inductiehypothese is de vervolgstand winnend voor de andere speler. Dit betekent dat het spel verliezend is voor de speler die begint, welke eerste zet d zij ook kiest.
Als $n = n_0 + 1$ is even, dan kan de speler die begint 1 lucifer weghalen, want 1 is uiteraard een deler van n en kleiner dan n. Dan blijven er $n - 1 = n_0$ lucifers over voor de andere speler, en dit aantal is oneven (want n is even). Volgens de inductiehypothese is deze vervolgtoestand verliezend voor de speler die aan de beurt is, de andere speler dus. Omdat de speler die begint er met haar eerste zet voor kan zorgen dat de andere speler verliest, is het spel in dit geval winnend voor haar

□

13:07.
13:51
2(a)

```
void hoogte (knoop *wortel)
{ int h;
  if (wortel != NULL)
  { h=0;
    wortel -> papa = NULL;    // niet nodig, want initieel al zo
    if (wortel -> links != NULL)
    { hoogte (wortel -> links);
      wortel -> links -> papa = wortel;
      h = wortel -> links -> hoogte + 1;
    }
    if (wortel -> rechts != NULL)
    { hoogte (wortel -> rechts);
      wortel -> rechts -> papa = wortel;
      if (wortel -> rechts -> hoogte >= h)
        h = wortel -> rechts -> hoogte + 1;
    }
    wortel -> hoogte = h;
  } // if wortel ≠ NULL
}
```

14.04

(b)

```
void toevoegen (knoop *wortel, knoop * nieuw)
{ knoop * loper;
  int lengte;
  loper = wortel;
  while (loper -> links != NULL && loper -> rechts != NULL)
  {   // twee kinderen
      if (loper -> links -> hoogte <= loper -> rechts -> hoogte)
          loper = loper -> links;
      else
          loper = loper -> rechts;
  }   // while

  nieuw -> links = NULL;
  nieuw -> rechts = NULL;
  nieuw -> papa = loper;
  nieuw -> hoogte = 0;
  if (loper -> links == NULL)
      loper -> links = nieuw;
  else
      loper -> rechts = nieuw;

  // nu nog, zo nodig, hoogte-velden aanpassen;
  // vergelijk hoogte-velden met lengte van pad naar nieuw
  lengte = 1;
  while (loper != NULL && loper -> hoogte < lengte)
  { loper -> hoogte = lengte;
    loper = loper -> papa;
    lengte ++;
  }   // while.

}   // toevoegen.
```

14:30

3 (a)

```
void reorganiseer1 (int A[], int n)
{ int i, j,
      tmp;

   i=0;    // even index
   j= n-1; // oneven index,
           // want n is 2-macht >= 2


   while (i<j)
   { tmp = A[i];    // verwissel A[i] en A[j]
     A[i] = A[j]
     A[j]= tmp
     i += 2;
     j -= 2;
   }
}
```

14.45

(b)

```
void reorganiseer2 (int A[], int i, int j)
{ int tmp;

   if (i<j)
   { tmp = A[i];
     A[i] = A[j];
     A[j] = tmp;
     reorganiseer2 (A, i+2, j-2)
   }
}
```

(c)

```
void reorganiseer3 (int A[], int links, int rechts)
                    // Pre: (rechts - links +1) is 2-macht >= 2
{ int m, i, j,
    tmp;

  if (rechts == links+1) // twee elementen
  { tmp = A[links];
    A[links] = A[rechts];
    A[rechts] = tmp;
  }
  else // minstens vier elementen
  { m = (links + rechts) /2;     // rechtergrens van linkerhelft
    reorganiseer3 (A, links, m);     // linker helft
    reorganiseer3 (A, m+1, rechts); // rechter helft
     // nu is: eerste kwart : negatief
     //        tweede kwart : positief
     //        derde kwart : negatief
     //        vierde kwart : positief
     // verwissel tweede en derde kwart
    i = (links + m) /2 + 1;    // begin van tweede kwart
    j = m+1;
    while (i<=m)
    { tmp = A[i];
      A[i] = A[j];
      A[j] = tmp;
      i++;
      j++;
    }
  } // else
} // reorganiseer3
```

(d)

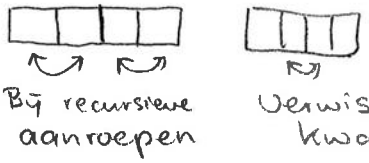
(i) n=4  : 1 verwisseling

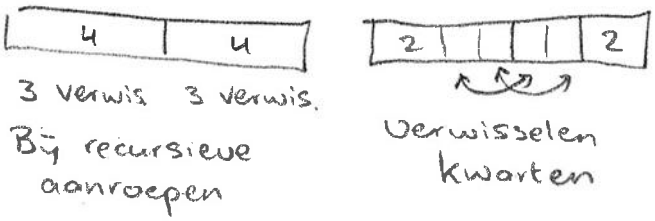
n=8  : 2 verwisselingen

n=16 : 4 verwisselingen

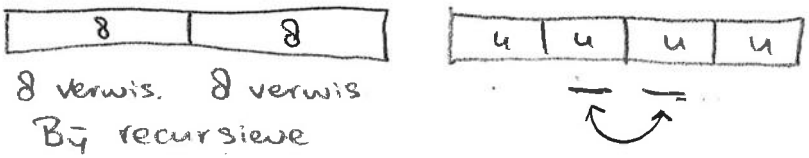
Algemene $n \geq 4$ (een 2-macht) : $\frac{1}{4}n$

(d)(ii)

n=4 :  : 3 verwisselingen

n=8 

: 3+3+2 = 8 verwisselingen.

n=16 

8+8+4 = 20 verwisselingen

15.55

4(a)

Ook best-fit-first branch and bound bouwt oplossingen component voor component op. Het houdt een verzameling van nog uit te werken deeloplossingen bij. Bij een minimalisatieprobleem wordt voor elke deeloplossing een ondergrens berekend voor de waarde van een complete oplossing die uit die deeloplossing voortkomt.

Ook nu wordt (de waarde van) de beste tot nu toe gevonden complete oplossing onthouden.

Iedere keer wordt uit de verzameling van nog uit te werken deeloplossingen één deeloplossing met de laagste ondergrens gepakt en helemaal uitgewerkt. Dat wil zeggen: voor elke mogelijkheid voor de volgende component wordt een nieuwe deeloplossing met eigen ondergrens geconstrueerd.

Een deeloplossing wordt gesnoeid (en dus niet meer uitgewerkt) als er geen geldige complete oplossing uit geconstrueerd kan worden of als de ondergrens hoger is dan of gelijk is aan de waarde van de beste tot nu toe gevonden complete oplossing.
Wanneer er uit een deeloplossing precies één complete oplossing geconstrueerd kan worden, wordt dat meteen gedaan.

En nog even voor de volledigheid:
* 'branch' is vertakken: we splitsen de complete toestandsruimte bij 'een' deeloplossing op in deelruimtes voor elke mogelijkheid voor de volgende component ('kinderen van de deeloplossing')
* 'bound' betekent dat we (bij minimalisatieproblemen) voor elke deeloplossing een ondergrens berekenen
* 'best-fit-first' betekent dat we steeds verder gaan met de deeloplossing met de laagste (= beste) ondergrens.

16.15

(b)
Belangrijke verschillen tussen backtracking en branch and bound zijn
* dat branch and bound alleen geschikt is voor optimalisatieproblemen terwijl backtracking ook geschikt is voor andere problemen
* dat branch and bound altijd met ondergrenzen (minimalisatie) of bovengrenzen (maximalisatie) werkt en backtracking niet per se
* dat branch and bound op basis van de ondergrens / bovengrens heen en weer kan springen in de toestandsruimte, terwijl backtracking een depth-first-search wandeling maakt.

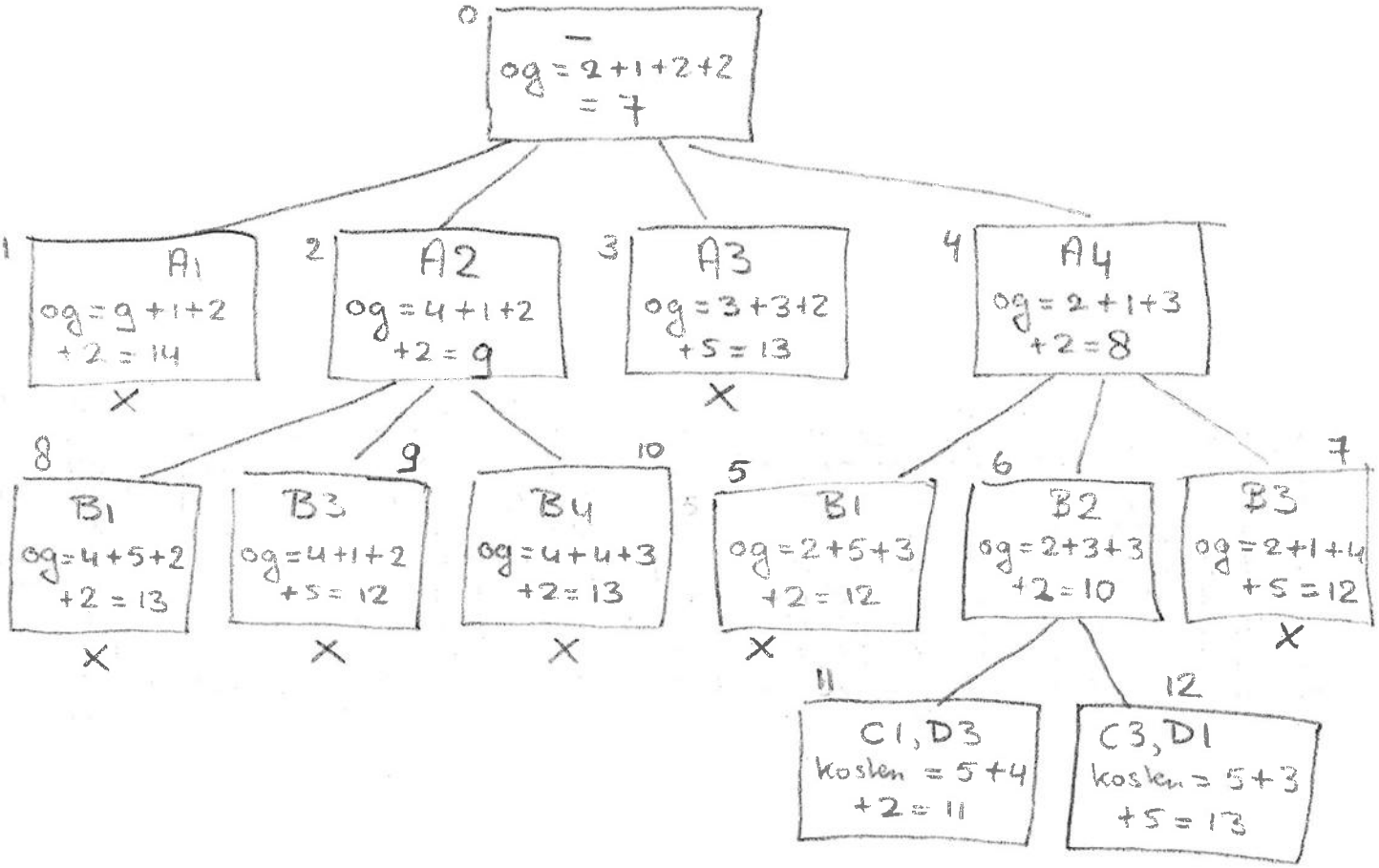
Branch and bound zal meestal sneller een minimale oplossing vinden, omdat het flexibeler door de toestandsruimte kan bewegen dan backtracking. Een deeloplossing kan er veelbelovend uitzien qua ondergrens, maar bij uitwerking toch minder gunstig blijken te zijn. Branch and bound kan dan verder gaan met een compleet andere deeloplossing, terwijl backtracking de eerder gekozen deeloplossing helemaal moet uitwerken.
En de kinderen van een gekozen deeloplossing worden bij backtracking doorgaans in een vaste volgorde afgelopen, terwijl branch and bound zich ook hier laat leiden door de ondergrens.

Doordat branch and bound vaak sneller een minimale oplossing vindt, kan het ook meer deeloplossingen snoeien, omdat hun ondergrens hoger dan of gelijk is aan de minimale waarde.

* dat branch and bound meerdere deeloplossingen tegelijk bijhoudt, verspreid over de toestandsboom en backtracking maar één. Branch and bound zoekt als het ware 'globaal'.

c] 14:09

In elke deeloplossing is een aantal ≥0 gebouwen al gekoppeld aan locaties. Dit levert al bepaalde kosten volgens de tabel: (x)
Voor elk van de resterende gebouwen kijken we in zijn rij wat de laagste kosten zijn bij nog ongebruikte locaties. Deze laagste waardes per resterend gebouw tellen we op bij de bepaalde kosten van (x).
Het resultaat is de ondergrens voor de deeloplossing



De deeloplossingen met X eronder worden gesnoeid, omdat hun ondergrens hoger (= slechter) is dan de waarde van de beste gevonden complete oplossing (11)

14.35

d] Een gretig algoritme kan eruit bestaan dat je voor elke rij in alfabetische volgorde de nog beschikbare locatie met de laagste kosten kiest
In ons voorbeeld levert dit gretige algoritme de volgende koppeling:

A4, B3, C1 (locaties 3 en 4 zijn niet meer beschikbaar), D2 (enige nog beschikbare locatie), met waarde
2 + 1 + 4 + 6 = 13

14.52.

Alternatief 1:

* kies voor elke locatie in oplopende volgorde het nog beschikbare gebouw met de laagste kosten

Alternatief 2:

* kies uit de hele tabel steeds de laagste waarde die mag (voor nog niet gekoppelde gebouwen en nog niet gekoppelde locaties.