1. (a) Merk op dat `func(n//16)` vier keer wordt aangeroepen (niet erg handig, maar het staat er nu eenmaal). Verder worden er 8 rekenkundige bewerkingen gebruikt (vier keer een vermenigvuldiging en een optelling), dus de recurrente betrekking wordt $T(n) = 4 \cdot T(n//16) + 8$.

   (b) Neem aan dat n een macht van 16 is (maakt voor het bepalen van de complexiteitsklasse niets uit), dan hebben we $T(n) = 4 \cdot T(n/16) + 8$. We passen nu het Master Theorema toe, met $a = 4$ en $b = 16$, dus $E = log4/log16 = 1/2$. Er geldt dat $8 \in O(n^{1/2-\epsilon})$ voor een ϵ, dus we hebben geval 1, dus $T(n) \in \Theta(n^{1/2})$.

2. Het grootste element van een maxheap heeft index 0. Swap dit met het laatste element, pas de lengte van de heap aan, en herstel de heapeigenschap middels *heapify*; dat laatste heeft een complexiteit $O(logn)$.

```
def delmax(E):
    i=E.heapsize-1
    E[0],E[i]=E[i],E[0]
    E.heapsize= E.heapsize-1
    heapify(E,0)
```

3. We zoeken eerst het grootste element op in de BST. We vinden die node met het maximum door netzolang naar rechts te gaan tot het niet meer kan.

   Vervolgens vinden we de voorganger ($\leq max$) door, als dat kan, naar links te gaan en vervolgens zoveel mogelijk naar rechts (daar zit het grootste element van de linkersubboom). Is er geen linkersubboom, dan is de ouder de voorganger (die ouder kan eventueel null zijn). Als de ouder niet null is, is de knoop met de grootste waarde een rechterkind van die ouder (dus de key van de ouder is kleiner of gelijk aan het maximum).

   Als het maximum zowel een linkersubboom heeft als een ouder, dan zijn alle keys van die linkersubboom groter dan de key van de ouder van het maximum (dus onze oplossing is in dat geval correct).

```
def een_na_grootste(T):
    x=T.root

    while x.right != null:
        x=x.right
```

```
if x.left == null:
    return x.parent
else:
    x=x.left
    while x.right != null:
        x=x.right
    return x
```

4. (a) Of je stopt voorwerp i niet in de rugzak, en dan is het restgewicht dus $R(i-1,g)$, of je stopt voorwerp i er wel in, en dan is het restgewicht $R(i-1,g-w_i)$. Hiervan moet je het minimum nemen, dus:
   $R(i,g) = min\{R(i-1,g), R(i-1,g-W_i)\}$.

   (b) Het algoritme om de matrix te vullen is als volgt (we gaan er hierbij vanuit dat `w` gevuld is vanaf `w[1]` tot en met `w[n]` en dus lengte n+1 heeft):

```
def backpack(w,G):

    n=len(w)-1
    R=[[0 for g in range(G+1)] for i in range(n+1)]

    for g in range(0,L+1):
        R[0,g]=g           # restgewicht gelijk aan g

    for i in range(1,n+1):
        for g in range(0,G+1):
            if (g-w[i])<0:
                R[i,g]=R[i-1,g]
            else:
                 R[i,g]=min(R[i-1,g],R[i-1,g-w[i]])

    return G-R[n,G];
```

   De complexiteit van dit algoritme is $\Theta(n^2)$.