Algorithm 1 Cuboid Reduction Method

procedure CRM(*droneAmount*, *dailyRequiremnt*)
    $ParameterOne$: The amount of drones needed for the container
    $ParameterTwo$: The daily requirement of Medical Packages $[MED1, MED2, MED3]$
    $Output$: The amount of days a container of supplies can last

    $cuboid \leftarrow [46, 46, 47]$ ▷ The dimensions of our cuboid
    $medPs \leftarrow [14, 7, 5; 5, 8, 5; 12, 7, 4]$ ▷ 2d Array of Medical Package Dimensions
    $cuboidsLeft \leftarrow 20 - droneAmount$ ▷ Amount of cuboids left after we pack in drones

    $packRatios \leftarrow RATIOCALCULATOR(cuboidsLeft, dailyRequirment, cuboid, medPs)$

    $packAmount \leftarrow [0, 0, 0]$
    for $i = 1$ to 3 **do**
        $packAmount[i] \leftarrow (INFITTER(cuboid, medPs[i]) * packRatios[i])$
    end for

    Return $DAYCALCULATOR(dailyRequirement, packAmount)$
end procedure

Algorithm 2 MED Pack cuboid Ratio Calculator

```
procedure RATIOCALCULATOR(cuboidAmt, dailyReq, cuboidDim, medDim)
    ParameterOne: The amount of Cuboids available
    ParameterTwo: The daily requirement of Medical Packages [MED1, MED2, MED3]
    ParameterThree: An array of dimensions of the cuboid
    ParameterFour: A 2d array of medical package dimensions
    Output: An array with the required amount of cuboids for each Medical Package

    ratios ← [0, 0, 0]
    order ← [0, 0, 0]

    total ← 0
    for i = 1 to 3 do
        total ← (total + dailyReq[i])
    end for

    for i = 1 to 3 do
        percentage ← (dailyReq[i]/total)
        tempRatio ← (percentage ∗ cuboidAmt)
        order[i] ← (tempRatio − floor(tempRatio))          ▷ track the percentage difference
        ratios[i] ← floor(tempRatio)
    end for

    total ← 0
    for i = 1 to 3 do
        total ← (total + ratios[i])
    end for

    cuboidsLeft ← (cuboidAmt − total)
    highestPriority ← order[i]
    place ← 1
    for i = 2 to 3 do ▷ find which med. pack. was most affected by the above process and give it priority
        if highestPriority < order[i] then
            highestPriority ← order[i]
            place ← i
        end if
    end for

    ratios(place) ← ratios(place) + cuboidsLeft          ▷ dedicate excess cuboid(s) to prioritised
    Return ratios
end procedure
```

Algorithm 3 'In-fitter' Algorithm

procedure INFITTER(box_1, box_2)
    $ParameterOne$: box_1 Dimensions of the bigger box (Array) $[L, W, H]$
    $ParameterTwo$: box_1 Dimensions of the smaller boxes (Array) $[L, W, H]$
    $Output$: The most amount of box_2 that will fit into box_1

    $perms \leftarrow [1, 2, 3; 1, 3, 2; 2, 1, 3; 2, 3, 1; 3, 1, 2; 3, 2, 1]$ ▷ 2d Array of box orientation
    $amountFit \leftarrow [6]$
    for $i = 1$ to 6 **do**
        $boxL \leftarrow box_2[perms[i][1]]$ ▷ get current permutation configuration
        $boxW \leftarrow box_2[perms[i][2]]$
        $boxH \leftarrow box_2[perms[i][3]]$

        $amountL \leftarrow floor(box_1[1]/boxL)$
        $amountW \leftarrow floor(box_1[2]/boxW)$
        $amountH \leftarrow floor(box_1[3]/boxH)$
        $amountFit[i] \leftarrow (amountL * amountW * amountH)$
    end for
    Return $findLargest(amountFit)$ ▷ return the largest number from the array
end procedure

Algorithm 4 Time Until Out of Supplies Calculator

procedure DAYCALCULATOR($dailyReq$, $packAmount$)
    $ParameterOne$: The daily requirement of Medical Packages $[MED1, MED2, MED3]$
    $ParameterTwo$: The amount of Medical Packages we have $[MED1, MED2, MED3]$
    $Output$: The amount of days the supply will last

    $days \leftarrow [0, 0, 0]$

    for $i = 1$ to 3 **do**
        $days[i] \leftarrow (packAmount[i]/dailyReq[i])$
    end for

    $mostDays \leftarrow days[1]$
    for $i = 2$ to 3 **do**
        if $mostDays < days[i]$ **then**
            $mostDays \leftarrow days[i]$
        end if
    end for

    Return $floor(mostDays)$
end procedure

Algorithm 5 'Out-fitter' Algorithm

procedure OUTFITTER(box_1, box_2)
    $ParameterOne$: box_1 Dimensions of the bigger box (Array) $[L, W, H]$
    $ParameterTwo$: box_1 Dimensions of the smaller boxes (Array) $[L, W, H]$
    $Output$: The least amount of box_2 that will be the same size or bigger than box_1

    $perms \leftarrow [1, 2, 3; 1, 3, 2; 2, 1, 3; 2, 3, 1; 3, 1, 2; 3, 2, 1]$ ▷ permutations of box orientation
    $amountFit \leftarrow [6]$
    for $i = 1$ to 6 **do**
        $boxL \leftarrow box_2[perms[i][1]]$ ▷ get current permutation configuration
        $boxW \leftarrow box_2[perms[i][2]]$
        $boxH \leftarrow box_2[perms[i][3]]$

        $amountL \leftarrow floor(box_1[1]/boxL)$
        $amountW \leftarrow floor(box_1[2]/boxW)$
        $amountH \leftarrow floor(box_1[3]/boxH)$

        if $remainder(box_1[1], boxL)$ **then** ▷ check to see if it fit exactly
            $amountL \leftarrow amountL + 1$ ▷ add another box to ensure full coverage
        end if
        if $remainder(box_1[2], boxL)$ **then**
            $amountW \leftarrow amountW + 1$
        end if
        if $remainder(box_1[3], boxL)$ **then**
            $amountH \leftarrow amountH + 1$
        end if

        $amountFit[i] \leftarrow (amountL * amountW * amountH)$
    end for
    Return $findSmallest(amountFit)$ ▷ return the smallest number from the array
end procedure