```julia
## Solver.jl

function init(n_jobs, n_processors, UB, duration, processor)

    s = [Tuple{Int,Int,Int,Int}[] for i in 1:n_processors]
    occupiedRanges = [Tuple{Int,Int}[] for i in 1:n_jobs]

    for operation in 1:n_processors
        for job in 1:n_jobs
            d = duration[job, operation]
            p = processor[job, operation]

            if (length(s[p]) == 0)
                newRange = (0,d)
            else
                prevElement = s[p][length(s[p])]
                newRange = (prevElement[4]+1,prevElement[4]+1+d)
            end
            overlappingRange = checkOverlap(newRange, occupiedRanges[job])
            while (!isnothing(overlappingRange))
                newRange = (overlappingRange[2]+1, overlappingRange[2]+1+d)
                overlappingRange = checkOverlap(newRange, occupiedRanges[job])
            end
            newElement = (job,operation,newRange[1],newRange[2])
            push!(s[p], newElement)

            insertElementInOccupiedRanges(occupiedRanges[job], newRange)
        end
    end
    return s, occupiedRanges
end

function randomStep(s, occupiedRanges, duration, processor, k)
    s = deepcopy(s)
    occupiedRanges = deepcopy(occupiedRanges)
    removedElements = NTuple{4, Int64}[]
    for i in 1:k
        randomProcIdx = rand(1:length(s))
        if (length(s[randomProcIdx]) > 0)
            randomElementIdx = rand(1:length(s[randomProcIdx]))
            removedElement = removeElement(s, occupiedRanges, randomProcIdx, randomElementIdx)
            insertElement(s, occupiedRanges, removedElement[1], removedElement[2], duration, processor)
            push!(removedElements, removedElement)
            s, occupiedRanges = compress(s, occupiedRanges, duration, randomProcIdx)
        end
    end
    return s, occupiedRanges
end

function compress(s, occupiedRanges, duration, procNum)
    s = deepcopy(s)
    occupiedRanges = deepcopy(occupiedRanges)
    processor = s[procNum]
    for i in eachindex(processor)
        element = processor[i]
        job = element[1]
        operation = element[2]
        startTime = element[3]
        endTime = element[4]

        newStartTime = 0
        newEndTime = newStartTime + duration[job, operation]

        if (i > 1)
            newStartTime = processor[i-1][4] + 1
            newEndTime = newStartTime + duration[job, operation]
        end

        or = occupiedRanges[job]
        occIdx = findfirst(x -> x == (startTime, endTime), or)
        if (occIdx > 1)
            if (or[occIdx-1][2] > newStartTime-1)
                newStartTime = or[occIdx-1][2] + 1
                newEndTime = newStartTime + duration[job, operation]
            end
```

```
        end

        newTuple = (processor[i][1], processor[i][2], newStartTime, newEndTime)
        processor[i] = newTuple
        or[occIdx] = (newStartTime, newEndTime)
    end
    return s, occupiedRanges
end

function removeElement(s, occupiedRanges, proc, index)
    processor = s[proc]
    job = processor[index][1]
    operation = processor[index][2]
    startTime = processor[index][3]
    endTime = processor[index][4]
    deleteat!(processor, index)

    occIdx = findfirst(x -> x == (startTime, endTime), occupiedRanges[job])
    deleteat!(occupiedRanges[job], occIdx)
    return (job, operation, startTime, endTime)
end

function insertElement(s, occupiedRanges, job, operation, duration, processor)
    d = duration[job, operation]
    p = processor[job, operation]

    if (length(s[p]) == 0)
        newRange = (0,d)
    else
        prevElement = s[p][length(s[p])]
        newRange = (prevElement[4]+1,prevElement[4]+1+d)
    end
    overlappingRange = checkOverlap(newRange, occupiedRanges[job])
    while (!isnothing(overlappingRange))
        newRange = (overlappingRange[2]+1, overlappingRange[2]+1+d)
        overlappingRange = checkOverlap(newRange, occupiedRanges[job])
    end
    newElement = (job,operation,newRange[1],newRange[2])
    push!(s[p], newElement)

    insertElementInOccupiedRanges(occupiedRanges[job], newRange)
end

# Inserts element using insertion sort
function insertElementInOccupiedRanges(ranges, element)
    i = 1
    while i <= length(ranges) && element > ranges[i]
        i += 1
    end
    insert!(ranges, i, element)
end

function checkOverlap(range, ranges)
    for i in ranges
        if (rangesOverlap(range, i))
            return i
        end
    end
    return nothing
end

function rangesOverlap(range1, range2)
    if (range1[1] > range2[2] || range1[2] < range2[1])
        return false
    end
    return true
end
```