```julia
### s194624.jl

using Random

include("IO.jl")
include("Solver.jl")

struct ArgumentException <: Exception
    message::String
end

function main()
    instanceLocation = ARGS[1]
    solutionLocation = ARGS[2]
    maxTime = parse(Int, ARGS[3])
    n_jobs, n_processors, UB, duration, processor, name = read_instance(instanceLocation)

    printInstance(n_jobs, n_processors, UB, duration, processor)

    println("Finding initial solution...")
    s, occupiedRanges = init(n_jobs, n_processors, UB, duration, processor)
    println("Initial solution found!")

    # Number of random elements to remove for neighbors
    k = 5

    elapsedTime = 0
    iterations = 0
    deltaSum = 0
    betaMinus = 0
    betaPlus = 0
    gamma = 0.9999999
    sCost = 0
    K = 10000
    T = 0

    # Rough assumption that number of iterations are linear to number of jobs (and that n_jobs = n_processors)
    # For example with 4 jobs the approximate number of iterations is about 103000, thus to generalize:
    estimatedIterationsPerSec = round(Int, (103000*4)/n_jobs)

    println("\nRunning simulated annealing algorithm for ", maxTime, " seconds...")
    start = time_ns()
    while (iterations <= K)
        if (iterations < K)
            sMark, occupiedRangesMark = randomStep(s, occupiedRanges, duration, processor, k)
            diff = Float64(cost(sMark) - cost(s))
            if (diff > 0)
                deltaSum += diff
                betaPlus += 1
            elseif (diff < 0)
                betaMinus += 1
            end
        else
            if (betaMinus == 0)
                betaMinus = 1
            end
            T = deltaSum / log(betaMinus / (betaMinus*gamma-betaPlus*(1-gamma)))
        end
        iterations += 1
    end

    eta = 0.00001
    estimatedIterations = estimatedIterationsPerSec*maxTime

    alpha = exp(log(-1/(T*log(eta)))/estimatedIterations)

    println("\nIteration ", K, ": Calculated initial temperature and temperature decay!")
    println("Initial temperature: ", T)
    println("Alpha decay: ", alpha)
    println()

    while (!terminate(elapsedTime, maxTime))
        sMark, occupiedRangesMark = randomStep(s, occupiedRanges, duration, processor, k)
        sMarkCost = cost(sMark)
        sCost = cost(s)
```

```julia
        delta = Float64(sMarkCost - sCost)

        if (iterations % round(Int, estimatedIterations / 15) == 0)
            println("Iteration ", iterations, "")
            println("Current objective value: ", sCost)
        end

        if (delta < 0 || rand() < exp(-(delta/T)))
            s = sMark
            occupiedRanges = occupiedRangesMark
        end

        #TODO Determine temperature decay
        T *= alpha

        elapsedTime = round((time_ns()-start)/1e9,digits=3)
        iterations += 1
    end

    println("\nTimed out")
    println(iterations, " actual iterations", )
    println(Int(round(estimatedIterations)), " estimated iterations")
    println()

    println("Final objective value: ", sCost)
    #printResults(s, occupiedRanges)

    if (ARGS[2] == " ")
        vals = rsplit(name, ".", limit=2)
        solutionLocation = string("sols/", vals[1], ".sol")
    end
    writeSolution(s, solutionLocation, n_jobs, n_processors)
end

function cost(s)
    max = 0
    for processor in s
        if (length(processor) > 0)
            element = processor[length(processor)]
            if (element[4] > max)
                max = element[4]
            end
        end
    end
    return max
end

function terminate(elapsedTime, maxTimeAllowed)
    if (elapsedTime > maxTimeAllowed)
        return true
    end
    return false
end

main()
```

```julia
## IO.jl

function read_instance(filename::String)
    f = open(filename)
    readline(f)#comment
    n_jobs, n_processors, UB = parse.(Int,split(readline(f)))
    readline(f)#comment
    duration = zeros(Int,n_jobs,n_processors)
    for i in 1:n_jobs
        duration[i,:] = parse.(Int,split(readline(f)))
    end
    readline(f)#comment
    processor = zeros(Int,n_jobs,n_processors)
    for i in 1:n_jobs
        processor[i,:] = parse.(Int,split(readline(f)))
    end
    close(f)

    path, file = split(filename, ".")
    dir, name = split(path, "/")
    return n_jobs, # the number of jobs
           n_processors, # the number of processors = number of operations
           UB, # the best-known upper bound
           duration, # the duration of each operation
           processor, # the processor assinged to each operation
           name
end

function writeSolution(solution, solutionLocation, n_jobs, n_processors)
    wDir = string(pwd())

    dir, file = splitdir(solutionLocation)
    if (!isdir(dir))
        mkpath(string("./", dir, "/"))
    end

    open(string(wDir, "/", solutionLocation), "w") do f
        for i in 1:n_jobs
            for j in 1:n_processors
                for processor in solution
                    for job in processor
                        if (i == job[1] && j == job[2])
                            write(f, string(job[3], " "))
                        end
                    end
                end
            end
            write(f, "\n")
        end
    end
end

function printInstance(n_jobs, n_processors, UB, duration, processor)
    println()
    println("Running instance with parameters:")
    println("Jobs: ", n_jobs)
    println("Processors: " , n_processors)
    println("duration: ", duration)
    println("processor ", processor)
    println()
end

function printResults(s, occupiedRanges)
    for i in eachindex(s)
        println("Processor ", i, ": ", s[i])
    end
    println()
    println("Occupied ranges: ")
    for i in eachindex(occupiedRanges)
        println("Job ", i, ": ", occupiedRanges[i])
    end
    println("Objective: ", cost(s))
end
```

```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
```