```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)
```

```
        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()
```