```julia
### s194624.jl

using Random

include("IO.jl")
include("ConstructionHeuristic.jl")
include("LocalSearch.jl")

struct ArgumentException <: Exception
    message::String
end

function main()
    ## Initialization
    instanceLocation = ARGS[1]
    solutionLocation = ARGS[2]
    maxTime = parse(Int, ARGS[3])

    if (length(ARGS) > 3)
        alpha = parse(Float32, ARGS[4])
    else
        alpha = 0.8
    end

    name, dim, LB , rev, rev_pair, k, H, p = read_instance(instanceLocation)
    printInstanceInformation(name, dim, LB , rev, rev_pair, k, H, p)

    if (ARGS[2] == " ")
        solutionLocation = string("sols/", name, ".sol")
    end

    bestKnownRev = 0
    bestKnownSol = [Int[] for i in 1:k]

    ## GRASP
    iterations = 0
    elapsedTime = 0
    println()
    println("Running GRASP for ", maxTime, " seconds...")
    println()
    start = time_ns()
    while (elapsedTime < maxTime)
        sol, revenue, mF = GRCPlast(dim, rev, rev_pair, k, H, p, alpha)
        newSol, newRevenue = LocalSearch(sol, revenue, dim, rev, rev_pair, k, H, p, alpha, mF, maxTime)
        if (newRevenue > bestKnownRev)
            println("Better solution found: ", newRevenue, " > ", bestKnownRev)
            bestKnownSol = newSol
            bestKnownRev = newRevenue
        end
        elapsedTime = round((time_ns()-start)/1e9,digits=3)
        iterations += 1
    end

    println()
    println("GRASP Executed...")
    println("Iterations: ", iterations)
    println("Final solution with revenue: ", bestKnownRev)
    println("Lower bound: ", LB)

    writeSolution(bestKnownSol, solutionLocation)
end
main()
```

```julia
### ConstructionHeuristic.jl

using Random

function GRCPlast(dim, rev, rev_pair, k, H, p, alpha)
    products = [i for i in 1:dim]
    mF = Int[0 for i in 1:k]
    sol = [Int[] for i in 1:k]

    # Put in initial elements to each assembly line
    totalRevenue = 0
    for i in eachindex(sol)
        # Extract random element
        element = rand(products)

        # Add to solution
        push!(sol[i], element)

        # Update revenue
        totalRevenue += rev[element]

        # Update available time
        mF[i] += p[element]

        # Mark as visited
        filter!(x -> x != element, products)
    end

    hasCandidates = [true for i in 1:k]
    while (true)
        for i in 1:k
            # Get candidate list
            candidates, revenues = getCandidates(sol[i], mF[i], products, rev, rev_pair, dim, alpha, H, p)
            if (!isempty(candidates))
                # Select element from candidate
                index = rand(1:length(candidates))
                element = candidates[index]

                # Add to solution
                append!(sol[i], element)

                # Update revenue
                totalRevenue += revenues[index]

                # Update available time
                mF[i] += p[element]

                # Mark as visited
                filter!(x -> x != element, products)
            else
                hasCandidates[i] = false
            end
        end
        if (!any(hasCandidates))
            break
        end
    end
    return sol, totalRevenue, mF
end

function getCandidates(productionLine, mF, products, rev, rev_pair, dim, alpha, H, p)
    revenues = fill(0, dim)
    for i in products
        revenues[i] += rev[i]
        for j in productionLine
            revenues[i] += rev_pair[i,j]
        end
    end
    nonNegativeValues = filter(x -> x >= 0, revenues)

    max = maximum(nonNegativeValues)
    min = Int(round(alpha*(max-minimum(nonNegativeValues))))

    candidates = findall(x -> withinRange(x, min, max), revenues)
    filter!(x -> p[x] + mF <= H, candidates)
```

```
    candidateValues = Int[]
    for i in candidates
        push!(candidateValues, revenues[i])
    end

    return candidates, candidateValues
end

function withinRange(x, a, b)
    return x >= a && x <= b
end
```

```julia
### IO.jl

function read_instance(filename)
    f = open(filename)
    name = readline(f) # name of the instance
    size = parse(Int32,readline(f)) # number of order
    LB = parse(Int32,readline(f)) # best known revenue
    rev = parse.(Int32,split(readline(f)))# revenue for including an order
    rev_pair = zeros(Int32,size,size) # pairwise revenues
    for i in 1:size-1
        data = parse.(Int32,split(readline(f)))
        j=i+1
        for d in data
            rev_pair[i,j]=d
            rev_pair[j,i]=d
            j+=1
        end
    end
    readline(f)
    k = parse(Int32,readline(f)) # number of production lines
    H = parse(Int32,readline(f)) # planning horizon
    p = parse.(Int32,split(readline(f))) # production times
    close(f)
    return name, size, LB ,rev, rev_pair, k, H, p
end

function printInstanceInformation(name, dim, LB , rev, rev_pair, k, H, p)
    println()
    println("Running instance with name: ", name)
    println("dim: ", dim)
    println("rev: ", typeof(rev), " size ", size(rev))
    println("rev_pair: ", typeof(rev_pair), " size ", size(rev_pair))
    println("k: ", k)
    println("H: ", H)
    println("p: ", typeof(p), " size ", size(p))
end

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

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

    open(string(wDir, "/", solutionLocation), "w") do f
        for i in eachindex(solution)
            for j in solution[i]
                write(f, string(j, " "))
            end
            write(f, "\n")
        end
    end
end
```

```julia
### LocalSearch.jl

function LocalSearch(sol, revenue, dim, rev, rev_pair, k, H, p, alpha, mF, maxTime)

    bestKnownSol = deepcopy(sol)
    bestKnownRev = deepcopy(revenue)
    bestKnownMF = deepcopy(mF)

    # Remove alpha
    elapsedTime = 0
    start = time_ns()

    while (elapsedTime < maxTime)
        newSol, newRevenue, newMF = swapImprovement(bestKnownSol, bestKnownRev, dim, rev, rev_pair, k, H, p, alpha, bestKnownMF)
        if (newRevenue > bestKnownRev)
            bestKnownSol = newSol
            bestKnownRev = newRevenue
            bestKnownMF = newMF
        else
            break
        end
        elapsedTime = round((time_ns()-start)/1e9,digits=3)
    end

    return bestKnownSol, bestKnownRev
end

# Uses first improvement
function swapImprovement(sol, revenue, dim, rev, rev_pair, k, H, p, alpha, mF)

    bestKnownSol = deepcopy(sol)
    bestKnownRev = deepcopy(revenue)
    bestKnownMF = deepcopy(mF)

    for i in 1:(k-1)
        for j in (i+1):k
            newSol, newRevenue, newMF = productionLineImprovement(bestKnownSol, bestKnownRev, dim, rev, rev_pair, k, H, p, alpha, i, j, bestKnownMF)
            if (newRevenue > bestKnownRev)
               return newSol, newRevenue, newMF
            end
        end
    end
    return bestKnownSol, bestKnownRev, bestKnownMF
end

function productionLineImprovement(sol, revenue, dim, rev, rev_pair, k, H, p, alpha, pl1, pl2, mF)

    bestKnownSol = deepcopy(sol)
    bestKnownRev = deepcopy(revenue)
    bestKnownMF = deepcopy(mF)

    for i in bestKnownSol[pl1]
        for j in bestKnownSol[pl2]
            if (legalPair(H, p, pl1, pl2, i, j, bestKnownMF))
                newSol, newRevenue, newMF = swap(bestKnownSol, bestKnownRev, dim, rev, rev_pair, k, H, p, alpha, pl1, pl2, i, j, bestKnownMF)
                if (newRevenue > bestKnownRev)
                    return newSol, newRevenue, newMF
                end
            end
        end
    end
    return bestKnownSol, bestKnownRev, bestKnownMF
end

function swap(sol, revenue, dim, rev, rev_pair, k, H, p, alpha, pl1, pl2, leftElement, rightElement, mF)

    finalSol = deepcopy(sol)
    finalRev = deepcopy(revenue)
    finalMF = deepcopy(mF)

    # Remove from each line
    filter!(x -> x != leftElement, finalSol[pl1])
    filter!(x -> x != rightElement, finalSol[pl2])

    # Update times
    finalMF[pl1] += (p[rightElement] - p[leftElement])
    finalMF[pl2] += (p[leftElement] - p[rightElement])

    # Update revenue
    for i in finalSol[pl1]
        finalRev -= rev_pair[leftElement, i]
        finalRev += rev_pair[rightElement, i]
    end
    for i in finalSol[pl2]
        finalRev -= rev_pair[rightElement, i]
        finalRev += rev_pair[leftElement, i]
    end

    # Insert back into lines
    push!(finalSol[pl1], rightElement)
    push!(finalSol[pl2], leftElement)

    return finalSol, finalRev, finalMF
end

function legalPair(H, p, pl1, pl2, leftElement, rightElement, mF)
    p1 = mF[pl1] + (p[rightElement] - p[leftElement]) <= H
```

```
    p2 = mF[pl2] + (p[leftElement] - p[rightElement]) <= H
    return p1 && p2
end
```