```julia
include("PlastOutReader.jl")

mutable struct solution
    objective::Float64 # total revenue
    order_mapping::Array{Int32, 2} # 2D array: rows = lines, cols = orders
end

mutable struct termination_criteria
    time_limit::UInt64
    not_improvement_limit::Int32
    candidate_solution::solution
    iteration::Int32
    not_improvement_count::Int32
    startTime::UInt64
end

function TimeElapsed(criteria::termination_criteria)
    return time_ns() - criteria.startTime
end

function Terminate(instance_data::problem_data, some_solution::solution, criteria::termination_criteria)
    criteria.iteration += 1

    if TimeElapsed(criteria) >= criteria.time_limit
        return true
    end

    if criteria.not_improvement_count >= criteria.not_improvement_limit
        return true
    end

    return false
end

function CalculateObjective(some_solution::solution, instance_data::problem_data)
    revenue = 0
    time = 0
    for line in 1:instance_data.no_prod_lines
        for order in 1:instance_data.no_orders
            if some_solution.order_mapping[line, order] != -1
                revenue += instance_data.revenue[order]
                time += instance_data.prod_time[order]
            end
        end
        # make sure the solution is feasible
        if time > instance_data.time_horizon
            return -Inf
        end
        time = 0
    end

    # if the solution is valid, we can add the cost savings
    for line in 1:instance_data.no_prod_lines
        for order_a in 1:instance_data.no_orders
            if some_solution.order_mapping[line, order_a] == -1
                continue
            end
            for order_b in order_a+1:instance_data.no_orders
                if some_solution.order_mapping[line, order_b] == -1
                    continue
                end
                revenue += instance_data.revenue_pair[order_a, order_b]
            end
        end
    end
    return revenue

end

function GreedyRandomizedConstruction(instance_data::problem_data, α::Float64)
    # attempts to find an initial fesible solution
    random_solution = solution(-Inf, fill(-1, instance_data.no_prod_lines, instance_data.no_orders))

    # initialize variables
    remaining_time = fill(instance_data.time_horizon, instance_data.no_prod_lines) # each production line has max time
    visited_orders = falses(instance_data.no_orders) # boolean to track visited orders

    # 1: select orders for production lines
    while any(.!visited_orders)
        # randomly select order to assign to a production line
        order = rand(1:instance_data.no_orders)
        if visited_orders[order]
            continue
        end

        # find feasible production lines for the random order
        candidate_lines = [] # list of production lines that can handle the order
        revenue_values = [] # list of total revenue values for each candidate line
        for line in 1:instance_data.no_prod_lines
            line_revenue = 0.0
            if remaining_time[line] >= instance_data.prod_time[order]
                # compute line revenue if order is assigned to this line
                line_revenue += instance_data.revenue[order] # profit for this order
                for assigned_order in 1:instance_data.no_orders # add cost savings from other orders
                    if random_solution.order_mapping[line, assigned_order] != -1 && assigned_order != order
                        line_revenue += instance_data.revenue_pair[order, assigned_order]
                    end
                end
                push!(candidate_lines, line)
                push!(revenue_values, line_revenue)
            end
        end

        if !isempty(candidate_lines)
            # calculate min/max revenue
            rMin = minimum(revenue_values)
            rMax = maximum(revenue_values)
```

```
            # calculate the RCL
            RCL = []
            for i in 1:length(candidate_lines)
                if revenue_values[i] >= rMin + α * (rMax - rMin)
                    push!(RCL, i)
                end
            end

            # randomly select a line from the RCL
            selected_line = RCL[rand(1:length(RCL))]

            # assign the order to the selected line
            random_solution.order_mapping[candidate_lines[selected_line], order] = order
            remaining_time[candidate_lines[selected_line]] -= instance_data.prod_time[order]
        end

        visited_orders[order] = true
    end
    random_solution.objective = CalculateObjective(random_solution, instance_data)
    return random_solution
end

function LocalSearch(instance_data::problem_data, some_solution::solution)
    improved = true

    while improved
        improved = false
        new_solution = solution(some_solution.objective, copy(some_solution.order_mapping))
        # move an order from one production line to another (if it increases the total revenue)
        for order in 1:instance_data.no_orders
            for assigned_line in 1:instance_data.no_prod_lines
                if some_solution.order_mapping[assigned_line, order] == -1 # get the assigned line
                    continue
                end

                for new_line in 1:instance_data.no_prod_lines # get an unused line
                    if new_line == assigned_line
                        continue
                    end

                    new_solution.order_mapping[new_line, order] = order
                    new_solution.order_mapping[assigned_line, order] = -1
                    new_solution.objective = CalculateObjective(new_solution, instance_data)

                    # if the new solution is better, update the solution
                    if new_solution.objective > CalculateObjective(some_solution, instance_data)
                        some_solution = new_solution
                        improved = true
                    end
                end
            end
        end
    end
    return some_solution
end

function StringRepresentation(some_solution::solution)
    return_string = ""
    for line in 1:size(some_solution.order_mapping, 1)
        for order in 1:size(some_solution.order_mapping, 2)
            if some_solution.order_mapping[line, order] != -1
                return_string *= string(some_solution.order_mapping[line, order]) * " "
            end
        end
        return_string *= "\n"
    end

    return return_string
end

function main()
    # receive user input and retrieve instance data
    if length(ARGS) < 3
        println("Usage: julia s242689.jl instances/<instance.txt> <solution_filename.txt> <time_limit_seconds>")
        exit(1)
    end
    instance_filename = ARGS[1]
    solution_filename = ARGS[2]
    time_limit_seconds = parse(Float64, ARGS[3])
    time_limit_ns = UInt64(time_limit_seconds * 1_000_000_000)
    problem_data = read_instance(instance_filename)

    if !isdir("sols")
        mkdir("sols")
    end

    # set up termination criteria and an empty current solution
    criteria = termination_criteria(time_limit_ns, 10000, solution(-Inf, fill(-1, problem_data.no_prod_lines, problem_data.no_orders)), 0, 0, time_ns())
    current_solution = solution(-Inf, fill(-1, problem_data.no_prod_lines, problem_data.no_orders))
    alpha = 0.3

    # GRASP algorithm
    while !Terminate(problem_data, current_solution, criteria)
        candidate_solution = GreedyRandomizedConstruction(problem_data, alpha) # randomized solution
        candidate_solution = LocalSearch(problem_data, candidate_solution) # perform local search around it

        if candidate_solution.objective > current_solution.objective
            current_solution = candidate_solution
            criteria.not_improvement_count = 0
        else
            criteria.not_improvement_count += 1
        end
    end

    # write the solution to a file
    f = open(solution_filename, "w")
    write(f, StringRepresentation(current_solution))
    close(f)
```

```
end

main()
```