1a)

Input: set of classes in list C_j with start time S_j and finish time $f_{j.}$

Output: all classes in list C scheduled in the least amount of classrooms.

Pseudo code:

```
C[$S_j$,$F_j$] // list of classes with start time and finish time
NumClasses = [] // number of classes being used
Sort C[$S_j$,$F_j$]
While length(C[$S_j$,$F_j$]) is not empty:
    choose C[$S_{x,}F_x$] with the lowest $S_x$
    if there is a class available for C[$S_{x,}F_x$] such that C[$S_x$] >= €{classroom[$F_x$]}. //
        place it in that classroom.
        Record next earliest spot of all classrooms AKA €{classroom[$F_x$]}.
    Else:
        Add classroom to NumClasses[].
        Add Class C[$S_{x,}F_x$]
        Record next earliest spot of all classrooms AKA €{classroom[$F_x$]}.
```

We want to schedule the most amount of classes with the least amount of rooms. To do this, we get all the classes and sort them by the starting point, Lowest to highest -- Big O(NlogN). We start with the least starting point and iterate thought the list -- (Big Θ(n)). We check if the starting time greater than or equal to the next earliest spot. If there is, we add it, if not we add a classroom and add the class to the new classroom. Record the next earliest spot Big O(1). Because the longest operation of this formula is to sort by the Greedy-Choice Property. The length of this equation is Big O(NLogN).

1b)

To minimize the penalties, we need to maximize the time to meet deadlines and work on the tasks with highest penalties.

Assumptions: each task has a cost >=1, each task has deadlines of whole minutes. There can be multiple of the same deadlines and costs.

Pseudo code:

```
T[$D_i$,$P_i$] // number of tasks
```

```
Sort by highest penalty in descending order

Slots[n] = {EMPTY} //Create time slots with N EMPTY intervals; n Being the largest Deadline.

While Slots[] are not EMPTY: // Fill with tasks; end when we reached SLOTS is full.

        TaskDeadline = min(n, T[D_i])

        If slot[taskDeadline] == EMPTY:

                Task[TaskDeadline] = task  // store task in schedule

        Else:

                  taskDeadline -= 1 //

                  while (taskDeadline != 0):  // check if other slots are available to store.

                        If slot[taskDeadline] == EMPTY:

                                Task[TaskDeadline] = task  // store task in schedule

                                break;

                        taskDeadline -= 1

append remaining tasks to schedule // this is the remaining tasks that will be penalized.
```

We want to prioritize the most penalized tasks. So sort the tasks by penalty in descending order. We know that we can only task N tasks – meaning there are the same number of slots as the task with the longest deadline. So we first attempt to schedule tasks according to the most expensive and place it in the spot where they can meet the deadline in the last minute. If that slot is full, we move down a slot and check its availability, we do that until we find a slot open and place it there. If there are no slots open, we moe on to the next task. We do this until we fill all slots in the schedule.

The cost to sort is (NlogN) if we attempt to use the merge sort algorithm with large datasets.

The cost to schedule tasks is O(N) since we will at most search the tasks once. However we can come up with the case that every other task can look for an open slot that fails to be added to the schedule, meaning we will have to search the entire schedule every other time: O((n^2)/2) => O(N^2)

Our run time will be O(N^2)

3)

Pseudo Code:

```
        T[$t_n$, $S_n$, $f_n$] // list of tasks [task number, Start time, End time]

        Sort  T[$t_n$, $S_n$, $f_n$] by $f_n$ in descending order

        taskList = [] // empty lost to add the task numbers
```

```
nextLatest = T[$s_0$] // start with the last task to finish
TaskList.prepend(T[$t_0$]) // add the first task to the tasks completed.
counter = 1
while ( NextLatest != 0 or numTask < Len(T[]))
    if T[$f_{counter}$] <= nextLatest
        nextLatest =  T[$S_{counter}$]
        TaskList.prepend(T[$t_{counter}$])
    numTask ++
output length of TaskList:
output TaskNum
```

This program sorts the tasks by the penalty size, in decreasing order. We then create a schedule with the largest deadline, since anything greater that what that tasksize can hold will be penalized. We then iterate through the sorted activities, we attempt to insert the slot at the time the activity is due. If thas slot is full in the schedule, we check the next slot that does not exceed the deadline. We do this until the task has been inserted, If no slot is open, we skip the task and are forced to take the penalty. We do this for all activities.

Runtime for scheduling (O(n^2))

Runtime for sorting (O(n)) // using selection sort.

This works because in each iteration, we can subdivide each task to schedule the task with the highest cost, based on the remaining activities available. The selecting factor being the penalty. Thus making a what could be Operation of (2^n) though a recusive method, we can reduce it to n^2, and even improve it to nlog(n) if we use the merge sort algorithm, since we are bounded by the sorting complexity of the program.