(a) Let J be the optimal subset. By definition all jobs in J can be scheduled to meet their deadline. Now consider the problem of scheduling to minimize the maximum lateness from class, but consider the jobs in J only. We know by the definition of J that the minimum lateness is 0 (i.e., all jobs can be scheduled in time), and in class we showed that the greedy algorithm of scheduling jobs in the order of their deadline, is optimal for minimizing maximum lateness. Hence ordering the jobs in J by the deadline generates a feasible schedule for this set of jobs.

(b) The problem is analogous to the Subset Sum Problem. We will have subproblems analogous to the subproblems for that problem. The first idea is to consider subproblems using a subset of jobs $\{1, \ldots, m\}$. As always we will order the jobs by increasing deadline, and we will assume that they are numbered this way, i.e., we have that $d_1 \leq \ldots \leq d_n = D$. To solve the original problem we consider two cases: either the last job n is accepted or it is rejected. If job n is rejected, then the problem reduces to the subproblem using only the first $n-1$ items. Now consider the case when job n is accepted. By part (a) we know that we may assume that job n will be scheduled last. In order to make sure that the machine can finish job n by its deadline D, all other jobs accepted by the schedule should be done by time $D - t_n$. We will define subproblems so that this problem is one of our subproblems.

For a time $0 \leq d \leq D$ and $m = 0, \ldots, n$ let $OPT(d, m)$ denote the maximum subset of requests in the set $\{1, \ldots, m\}$ that can be satisfied by the deadline d. What we mean is that in this subproblem the machine is no longer available after time d, so all requests either have to be scheduled to be done by deadline d, or should be rejected (even if the deadline d_i of the job is $d_i > d$). Now we have the following statement.

(1)

- *If job m* **is not** *in the optimal solution $OPT(d, m)$ then $OPT(m, d) = OPT(m-1, d)$.*

- *If job m* **is** *in the optimal solution $OPT(m, d)$ then $OPT(m, d) = OPT(m-1, d-t_m) + 1$.*

This suggests the following way to build up values for the subproblems.

```
Select-Jobs(n,D)
  Array M[0...n, 0...D]
  Array S[0...n, 0...D]
  For d = 0, ..., D
     M[0, d] = 0
     S[0, d] = φ
   Endfor
   For m = 1, ..., n
     For d = 0, ..., D
       If M[m-1, d] > M[m-1, d-t_m] + 1 then
```

[1] ex601.300.669

```
        M[m, d] = M[m − 1, d]
        S[m, d] = S[m − 1, d]
      Else
        M[m, d] = M[m − 1, d − t_m] + 1
        S[m, d] = S[m − 1, d − t_m] ∪ {m}
      Endif
    Endfor
  Endfor
  Return M[n, D] and S[n, D]
```

The correctness follows immediately from the statement (1). The running time of $O(n^2D)$ is also immediate from the for loops in the problem, there are two nested `for` loops for m and one for d. This means that the internal part of the loop gets invoked $O(nD)$ time. The internal part of this `for` loop takes $O(n)$ time, as we explicitly maintain the optimal solutions. The running time can be improved to $O(nD)$ by not maintaining the S array, and only recovering the solution later, once the values are known.