```
1   Algorithm GreedyJob(d, J, n)
2   // J is a set of jobs that can be completed by their deadlines.
3   {
4       J := {1};
5       for i := 2 to n do
6       {
7           if (all jobs in J ∪ {i} can be completed
8               by their deadlines) then J := J ∪ {i};
9       }
10  }
```

Algorithm 4.5 High-level description of job sequencing algorithm

$\ldots$, $J[k]$ are changed after the insertion. Hence, it is necessary to verify only that these jobs (and also job i) do not violate their deadlines following the insertion. The algorithm that results from this discussion is function JS (Algorithm 4.6). The algorithm assumes that the jobs are already sorted such that $p_1 \geq p_2 \geq \cdots \geq p_n$. Further it assumes that $n \geq 1$ and the deadline $d[i]$ of job i is at least 1. Note that no job with $d[i] < 1$ can ever be finished by its deadline. Theorem 4.5 proves that JS is a correct implementation of the greedy strategy.

Theorem 4.5 Function JS is a correct implementation of the greedy-based method described above.

Proof: Since $d[i] \geq 1$, the job with the largest p_i will always be in the greedy solution. As the jobs are in nonincreasing order of the p_i's, line 8 in Algorithm 4.6 includes the job with largest p_i. The **for** loop of line 10 considers the remaining jobs in the order required by the greedy method described earlier. At all times, the set of jobs already included in the solution is maintained in J. If $J[i]$, $1 \leq i \leq k$, is the set already included, then J is such that $d[J[i]] \leq d[J[i+1]]$, $1 \leq i < k$. This allows for easy application of the feasibility test of Theorem 4.3. When job i is being considered, the **while** loop of line 15 determines where in J this job has to be inserted. The use of a fictitious job 0 (line 7) allows easy insertion into position 1. Let w be such that $d[J[w]] \leq d[i]$ and $d[J[q]] > d[i]$, $w < q \leq k$. If job i is included into J, then jobs $J[q]$, $w < q \leq k$, have to be moved one position up in J (line 19). From Theorem 4.3, it follows that such a move retains feasibility of J iff $d[J[q]] \neq q$, $w < q \leq k$. This condition is verified in line 15. In addition, i can be inserted at position $w+1$ iff $d[i] > w$. This is verified in line 16 (note $r = w$ on exit from the **while** loop if $d[J[q]] \neq q$, $w < q \leq k$). The correctness of JS follows from these observations. □

```
1   Algorithm JS(d, j, n)
2   // d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
3   // are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
4   // is the ith job in the optimal solution, 1 ≤ i ≤ k.
5   // Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
6   {
7       d[0] := J[0] := 0; // Initialize.
8       J[1] := 1; // Include job 1.
9       k := 1;
10      for i := 2 to n do
11      {
12          // Consider jobs in nonincreasing order of p[i]. Find
13          // position for i and check feasibility of insertion.
14          r := k;
15          while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
16          if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
17          {
18              // Insert i into J[ ].
19              for q := k to (r + 1) step −1 do J[q + 1] := J[q];
20              J[r + 1] := i; k := k + 1;
21          }
22      }
23      return k;
24  }
```

Algorithm 4.6 Greedy algorithm for sequencing unit time jobs with deadlines and profits

For JS there are two possible parameters in terms of which its complexity can be measured. We can use n, the number of jobs, and s, the number of jobs included in the solution J. The **while** loop of line 15 in Algorithm 4.6 is iterated at most k times. Each iteration takes $\Theta(1)$ time. If the conditional of line 16 is true, then lines 19 and 20 are executed. These lines require $\Theta(k - r)$ time to insert job i. Hence, the total time for each iteration of the **for** loop of line 10 is $\Theta(k)$. This loop is iterated $n - 1$ times. If s is the final value of k, that is, s is the number of jobs in the final solution, then the total time needed by algorithm JS is $\Theta(sn)$. Since $s \leq n$, the worst-case time, as a function of n alone is $\Theta(n^2)$. If we consider the job set $p_i = d_i = n - i + 1$, $1 \leq i \leq n$, then algorithm JS takes $\Theta(n^2)$ time to determine J. Hence, the worst-case computing time for JS is $\Theta(n^2)$. In addition to the space needed for d, JS needs $\Theta(s)$ amount of space for J.