An optimal algorithm is to schedule the jobs in decreasing order of w_i/t_i. We prove the optimality of this algorithm by an exchange argument.

Thus, consider any other schedule. As is standard in exchange arguments, we observe that this schedule must contain an *inversion* — a pair of jobs i, j for which i comes before j in the alternate solution, and j comes before i in the greedy solution. Further, in fact, there must be an adjacent such pair i, j. Note that for this pair, we have $w_j/t_j \geq w_i/t_i$, by the definition of the greedy schedule. If we can show that swapping this pair i, j does not increase the weighted sum of completion times, then we can iteratively do this until there are no more inversions, arriving at the greedy schedule without having increased the function we're trying to minimize. It will then follow that the greedy algorithm is optimal.

So consider the effect of swapping i and j. The completion times of all other jobs remain the same. Suppose the completion time of the job before i and j is C. Then before the swap, the contribution of i and j to the total sum was $w_i(C + t_i) + w_j(C + t_i + t_j)$, while after the sum it is $w_j(C + t_j) + w_i(C + t_i + t_j)$. The difference between the value after the swap, compared to the value before the swap is (canceling terms in common between the two expressions) $w_i t_j - w_j t_i$. Since $w_j/t_j \geq w_i/t_i$, this difference is bounded above by 0, and so the total weighted sum of completion times does not increase due to the swap, as desired.

[1] ex948.540.252

Let c_e denote the cost of the edge e and we will overload the notation and write c_{st} to denote the cost of the edge between the nodes s and t.

This problem is by its nature quite similar to the shortest path problem. Let us consider a two-parameter function $Opt(i, s)$ denoting the optimal cost of shortest path to s using *exactly* i edges, and let $N(i, s)$ denote the number of such paths.

We start by setting $Opt(i, v) = 0$ and $Opt(i, v') = \infty$ for all $v' \neq v$. Also set $N(i, v) = 1$ and $N(i, v') = 0$ for all $v' \neq v$. Intuitively this means that the source v is reachable with cost 0 and there is currently one path to achieve this.

Then we compute the following recurrence:

$$Opt(i, s) = min_{t,(t,s) \in E}\{Opt(i-1, t) + c_{ts}\}. \tag{1}$$

The above recurrence means that in order to travel to node s using exactly i edges, we must travel a predecessor node t using exactly $i - 1$ edges and then take the edge connecting t to s. Once of course the optimal cost value has been computed, the number of paths that achieve this optimum would be computed by the following recurrence:

$$N(i, s) = \sum_{t,(t,s) \in E \ \ and \ \ Opt(i,s) = Opt(i-1,t) + c_{ts}} N(i-1, t). \tag{2}$$

In other words, we look at all the predecessors from which the optimal cost path may be achieved and add all the counters.

The above recurrences can be calculated by a double loop, where the outside loops over i and the inside loops over all the possible nodes s. Once the recurrences have been solved, our target optimal path to w is obtained by taking the minimum of all the paths of different lengths to w - that is:

$$Opt(w) = min_i\{Opt(i, w)\}. \tag{3}$$

And the number of such paths can be computed by adding up the counters of all the paths which achieve the minimal cost.

$$N(w) = \sum_{i, Opt(i,w) = Opt(w)} N(i, w). \tag{4}$$

[1] ex720.859.203

The basic idea is to ask: How should we gerrymander precincts 1 through j, for each j? To make this work, though, we have to keep track of a few extra things, by adding some variables. For brevity, we say that the A-votes in a precinct are the votes for party A, and B-votes are the votes for party B. We keep track of the following information about a partial solution.

- How many precincts have been assigned to district 1 so far?
- How many A-votes are in district 1 so far?
- how many A-votes are in district 2 so far?

So let $M[j, p, x, y] = true$ if it is possible to achieve at least x A-votes in district 1 and y A-votes in district 2, while allocating p of the first j precincts to district 1. ($M[j, p, x, y] = false$ otherwise.) Now suppose precinct $j + 1$ has z A-votes. To compute $M[j + 1, p, x, y]$, you either put precinct $j + 1$ in district 1 (in which case you check the results of sub-problem $M[j, p - 1, x - z, y]$) or in precinct 2 (in which case you check the results of sub-problem $M[j, p, x, y - z]$). Now to decide if there's a solution to the whole problem, you scan the entire table at the end, looking for a value of "true" in any entry of the form $M[n, n/2, x, y]$, where each of x and y is greater than $mn/4$. (Since each district gets $mn/2$ votes total.)

We can build this up in order of increasing j, and each sub-problem takes constant time to compute, using the values of smaller sub-problems. Since there are n^2m^2 sub-problems, the running time is $O(n^2m^2)$.

[1]ex706.269.18

The subproblems will represent the optimum way to satisfy orders $1, \ldots, i$ with an inventory of s trucks left over after the month i. Let $OPT(i, s)$ denote the value of the optimal solution for this subproblem.

- The problem we want to solve is $OPT(n, 0)$ as we do not need any leftover inventory at the end.
- The number of subproblems is $n(S + 1)$ as there could be $0, 1, \ldots, S$ trucks left over after a period.
- To get the solution for a subproblem $OPT(i, s)$ given the values of the previous subproblems, we have to try every possible number of trucks that could have been left over after the previous period. If the previous period had z trucks left over, then so far we paid $OPT(i - 1, z)$ and now we have to pay zC for storage. In order to satisfy the demand of d_i and have s trucks left over, we need $s + d_i$ trucks. If $z < s + d_i$ we have to order more, and pay the ordering fee of K.

  In summary the cost $OPT(i, s)$ is obtained by taking the smaller of $OPT(i-1, s+d_i)+C(s+d_i)$ (if $s+d_i \leq S$), and the minimum over smaller values of z, $\min_{z<\min(s+d_i,S)}(OPT(i-1, z) + zC + K)$.

  We can also observe that the minimum in this second term is obtained when $z = 0$ (if we have to reorder anyhow, why pay storage for any extra trucks?). With this extra observation we get that

  – if $s + d_i > S$ then $OPT(i, s) = OPT(i - 1, 0) + K$,

  – else $OPT(i, s) = \min(OPT(i - 1, s + d_i) + C(s + d_i), OPT(i - 1, 0) + K)$.

[1] ex304.359.690

(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 $M[0 \ldots n, 0 \ldots D]$
    Array $S[0 \ldots n, 0 \ldots D]$
    For $d = 0, \ldots, D$
        $M[0, d] = 0$
        $S[0, d] - \phi$
     Endfor
     For $m = 1, \ldots, n$
        For $d = 0, \ldots, D$
           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
            $M[m,d] = M[m-1,d-t_m] + 1$
            $S[m,d] = S[m-1,d-t_m] \cup \{m\}$
        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.

We build the following flow network. There is a node v_i for each client i, a node w_j for each base station j, and an edge (v_i, w_j) of capacity 1 if client i is within range of base station j. We then connect a super-source s to each of the client nodes by an edge of capacity 1, and we connect each of the base station nodes to a super-sink t by an edge of capacity L.

We claim that there is a feasible way to connect all clients to base stations if and only if there is an s-t flow of value n. If there is a feasible connection, then we send one unit of flow from s to t along each of the paths s, v_i, w_j, t, where client i is connected to base station j. This does not violate the capacity conditions, in particular on the edges (w_j, t), due to the load constraints. Conversely, if there is a flow of value n, then there is one with integer values. We connect client i to base station j if the edge (v_i, w_j) carries one unit of flow, and we observe that the capacity condition ensures that no base station is overloaded.

The running is the time required to solve a max-flow problem on a graph with $O(n+k)$ nodes and $O(nk)$ edges.

[1] ex751.45.676

We will assume that the flow f is integer-valued. Let $e^* = (v, w)$. If the edge e^* is not saturated with flow, then reducing its capacity by one unit does not cause a problem. So assume it is saturated.

We first reduce the flow on e^* to satisfy the capacity conditions. We now have to restore the capacity conditions. We construct a path from w to t such that all edges carry flow, and we reduce the flow by one unit on each of these edges. We then do the same thing from v back to s. Note that because the flow f is acyclic, we do not encounter any edge twice in this process, so all edges we traverse have their flow reduced by exactly one unit, and the capacity condition is restored.

Let f' be the current flow. We have to decide whether f' is a maximum flow, or whether the flow value can be increased. Since f was a maximum flow, and the value of f' is only one unit less than f, we attempt to find a single augmenting path from s to t in the residual graph $G_{f'}$. If we fail to find one, then f' is maximum. Else, the flow is augmented to have a value at least that of f; since the current flow network cannot have a larger maximum flow value than the original one, this is a maximum flow.

[1]ex257.178.863

If the minimum s-t cut has size $\leq k$, then we can reduce the flow to 0. Otherwise, let $f > k$ be the value of the maximum s-t flow. We identify a minimum s-t cut (A, B), and delete k of the edges out of A. The resulting subgraph has a maximum flow value of at most $f - k$.

But we claim that for any set of edges F of size k, the subgraph $G' = (V, E - F)$ has an s-t flow of value at least $f - k$. Indeed, consider any cut (A, B) of G'. There are at least f edges out of A in G, and at most k have been deleted, so there are at least $f - k$ edges out of A in G'. Thus, the minimum cut in G' has value at least $f - k$, and so there is a flow of at least this value.

[1] ex225.750.725

(a) Let $G = (V, E)$ be a bipartite graph with a node $p_i \in V$ representing each person and a node $d_j \in V$ representing each night. The edges consist of all pairs (p_i, d_j) for which $d_j \notin S_i$.

Now, a perfect matching in G is a pairing of people and nights so that each person is paired with exactly one night, no two people are paired with the same night, and each person is available on the night they are paired with. Thus, if G has a perfect matching, then it has a feasible dinner schedule. Conversely, if G has a feasible dinner schedule, consisting of a set of pairs $S = \{(p_i, d_j)\}$, then each $(p_i, d_j) \in S$ is an edge of G, no two of these pairs share an endpoint in G, and hence these edges define a perfect matching in G.

(b) An algorithm is as follows. First, construct the bipartite graph G from part (a): it takes $O(1)$ time to decide for each pair (p_i, d_j) whether it should be an edge in G, so the total time to construct G is $O(n^2)$. Now, let Q denote the set of edges constructed by Alanis. Delete the edge (p_j, d_k) from Q, obtaining a set of edges Q'. Q' has size $n - 1$, and since no person or night appears more than once in Q', it is a matching.

We now try to find an augmenting path in G with respect to Q', in time $O(|E|) = O(n^2)$. If we find such an augmenting path P, then increasing Q' using P gives us a matching of size n, which is a perfect matching and hence by (a) corresponds to a feasible dinner schedule. If G has no augmenting path with respect to Q', then by a theorem from class, Q' must be a maximum matching in G. In particular, this means that G has no perfect matching, and hence by (a) there is no feasible dinner schedule.

[1] ex245.875.267

(a) Define a flow network as follows. There is a source s, a node x_i representing each balloon i, a node z_i representing each condition c_i, and a sink t. There are edges (s, x_i) of capacity 2, (x_i, z_j) of capacity 1 whenever $c_j \in S_i$, and edges (z_j, t) of capacity k. We then test whether the maximum s-t flow has value nk.

The Ford-Fulkerson algorithm to find a maximum flow has running time $O(|E|C)$, where $|E|$ is the number of edges and C is the total capacity of edges out of s. Here we have $|E| = O(mn)$ and $C = 2m$, so the running time is $O(m^2n)$.

An alternate but related solution would place a lower bound of k on each edge of the form (z_j, t). One would then place a demand of $-nk$ on s and nk on t, and test whether there is a feasible circulation.

(b) Break all edges (x_i, z_j). Insert new nodes y_{pj} for each sub-contractor p and condition c_j. Add an edge (x_i, y_{pj}) of capacity 1 when $c_j \in S_i$ and balloon i is produced by sub-contractor p. Add an edge (y_{pj}, c_j) of capacity $k - 1$. Again, test whether the maximum s-t flow has value nk.

As in part (a), the running time is $O(|E|C)$. Here $|E|$ is still $O(mn)$, and $C = 2m$, so the running time is $O(m^2n)$.

The alternate solution in (a) based on circulations with lower bounds can be modified in the same way.

[1] ex557.796.690