Problem 1. (a) The basic idea of the greedy algorithm is to iterate through the edges and add any edge that does not conflict with previous choices to the solution. This can be implemented in linear time as follows:

- Initialize a boolean array `deleted` of length n with all values set to `False` to keep track of which nodes have been deleted from the graph.
- Iterate over the m edges one by one.
  - For each edge (u, v), if `deleted[u] == False` and `deleted[v] == False`, then we can safely add the edge to the solution, and set `deleted[u]` and `deleted[v]` to `True`.
  - Otherwise, if either u or v have been previously deleted, it means that that node has already been matched to a previous edge, so the current edge cannot be added to the solution.
- Return the set of edges in the solution.

This algorithm takes $O(n)$ time to initialize the array and $O(m)$ to iterate over the edges, so the overall runtime is linear in $O(n + m)$.

Next, we show that the greedy algorithm is a 2-approximation algorithm. Consider OPT, in which the maximum bipartite matching is found. Each time the greedy algorithm adds an edge to its solution, it marks two nodes u, v to be deleted, which can correspond to up to 2 edges of OPT being deleted (one edge with an endpoint at u, and one edge with an endpoint at v). Thus, after adding all possible edges, the greedy algorithm is guaranteed to have at least $\frac{1}{2}|OPT|$ edges in its solution, which makes it a valid 2-approximation algorithm.

(b) To generalize the previous algorithm to the positive-weight case, rather than iterating over the entire list of edges, we can instead consider one node at a time: for each node u, sort the list of outgoing edges (u, v_i) in decreasing order of weight and greedily select the outgoing edge with the greatest weight that does not conflict with any previous choices. Note that since each node needs only sort a subset of edges, the overall sorting time for all nodes is $O(n \log n)$. Thus, including initialization, the overall runtime of this greedy algorithm is $O(m \log n)$.

To show that this greedy algorithm is a 2-approximation for maximum weight bipartite matching, consider that for each edge (u, v) with weight w selected by the greedy algorithm, there are at most two corresponding edges in OPT (one edge with an endpoint at u and one edge with an endpoint at v with weights w'_1, w'_2 respectively). However, since the greedy algorithm always chooses the maximum weight edge for each vertex, we know that $w'_1 + w'_2 \leq 2w$. It follows that the total weight of the solution found by the greedy algorithm is at least $\frac{1}{2}|OPT|$.

(c) The algorithm from part (b) still works in the general case, since the bipartite restriction does not affect how the edges are selected. In other words, we still iterate over the list of nodes, and for each node u, we sort the set of outgoing edges (u, v_i) by order of weight, and greedily determine which edge to accept before marking u, v_i to be deleted. As with part (b), for each edge that we choose in the greedy algorithm, there are two corresponding edges in OPT with sum at most $2w$, so the greedy algorithm is still a valid 2-approximation for the general case.

Problem 2.

Problem 3. (a) Suppose we have a feasible subset of jobs that is not in order of increasing deadline. Assume that there are two adjacent jobs j, i, where j is scheduled before i, but d_j is after d_i (i.e. $d_j \geq d_i$). Let t_i be the time at which i finishes. If we swap the positions of j and i, the new finishing time of j will be t_i, which is feasible ($\leq d_i$), and therefore also $\leq d_j$ by definition. Thus, swapping j and i retains the feasibility of the subset. Therefore, we can continuously swap out-of-order jobs i, j in the subset until the jobs are in order of increasing deadline, while still being feasible. It follows that any feasible subset of jobs can be ordered by increasing deadline.

(b) From part (a), given any feasible subset of jobs S, we can schedule the subset in order of increasing deadline (i.e. as long as we have a set that is feasible, we don't need to worry about the order in which the jobs are scheduled). Then, we can construct a DP matrix `dp`, where `dp[i][j]` stores the fastest-completing subset of jobs $1, 2, \ldots, i$ with weight at least j. We can populate the DP matrix according to the following recursion:

$$\texttt{dp[i][j]} = \begin{cases} \infty & \text{if } i = 0, j \neq 0 \\ 0 & \text{if } j = 0 \\ \texttt{dp[i-1][j]} & \text{if } d_i < \texttt{dp[i-1][j - w\_i]} + p_i \\ \min\{\texttt{dp[i-1][j - w\_i]} + p_i, \texttt{dp[i-1][j]}\} & \text{if } d_i \geq \texttt{dp[i-1][j - w\_i]} + p_i \end{cases}$$

In other words, we initialize the initial total processing times to be ∞ (except if $j = 0$) and iteratively consider whether scheduling job i (if it is possible to be scheduled before the daedline) will lower the total weight of the subset. After constructing the DP matrix, the fastest-completing maximum-weight feasible subset is will therefore be the subset corresponding to `dp[n][W]`, where $W = \sum w_i$.

This algorithm runs in polynomial time, since the weights w_j are poynomial in n, and therefore constructing the DP matrix will take $O(\text{poly}(n))$ time.

Problem 4. (a) If there are only k distinct item sizes, then we can represent the input as an array of length k, where $arr[i]$ refers to the number of i-th smallest items. If we also define P to be the set of item packings that can fit into a single bin, then we can formulate the following DP recursion:

$$dp(arr[1,\ldots,k]) = \min_{p \in P} dp(arr[1-p_1,\ldots,k-p_k]) + 1$$

In other words, we can build up the DP matrix by trying to introduce bins one at a time, and minimizing the assignment of items outside the new bin each time. Thus, if we initialize the DP recursion for $dp([0,\ldots,0])$ and iteratively build up the solution until we reach the input, we will have determined the minimum assignment of bins.

The DP matrix is a $n \times k$ matrix, and for each slot, we must calculate the minimum of n^k possibilities (since there are n^k possible single-item packings in P). Thus, the overall runtime of the algorithm is polynomial.

(b) Suppose in the worst case that the remaining items are all size ϵ. We can continuously iterate over the bins until we have found a bin with remaining space at least ϵ, and repeat this process until we have run out of items to place. If there are no remaining bins with space at least ϵ, we will provision a new bin instead. For each new item, since we iterate over a progressively smaller set of bins, the runtime is linear.

Thus, we either use B bins (in the event that all of the existing bins have enough space to accommodate the remaining small items), or we will end up using $x > B$ bins. Since the total size of all items is at most B^*, and by construction we know that there are at least $x-1$ bins with at least $1-\epsilon$ size used, we have that the size of the placed items is bounded by $(x-1)(1-\epsilon) < B^*$, which implies $x < 1 + \frac{B^*}{1-\epsilon} < 1 + (1+2\epsilon)B^*$. Thus, the total number of bins used is $\max(B, 1 + (1+2\epsilon)B^*)$.

(c) We cannot round to the next integer power of $(1+\epsilon)$ due to the unit size restriction of each bin. For instance, consider the example where there are 10 items, each with size $1/2$. Then the optimal number of bins is 5, but if we increase the sizes of each item to some power of $(1+\epsilon)$, then the new optimum is at least 10, which is a two-fold increase at minimum (even if ϵ is small). Moreover, any item that is scaled to have greater than 1 size will render the problem impossible, since it cannot fit into any bin at that point.

(d) Assume that to start with, n items fit into B bins. Now, order the items by size and split them into k groups, and scale the size of each item to equal the maximum of its group. If we set aside S_1 for now, it is easy to see that the remaining items will all fit into the original B bins, since for each item in each group, we know by definition that its new size is still strictly less than the smallest size of the next largest group (because the maximum of any group is still ordered after the minimum of the preceding group), and therefore each item can take the place of an item in the next group, i.e. the items in S_2 will occupy the former positions of the items in S_1. Then all that remains is to place the items in S_1 that we set aside earlier. In we will have to use n/k new bins, one for each item. Thus, the grouping procedure increases the number of bins used by at most n/k.

(e) We can combine the results of the previous parts to create an algorithm as follows:

- We will first attempt to pack all items of size greater than $\epsilon/2$ into B bins as in part (b), where we want B to be at most $1 + (1+\epsilon)B^*$. Suppose there are x items greater

than $\epsilon/2$. To pack x into B, we use the grouping procedure as in part (d), with $k = 2/\epsilon^2$ (since x is bounded by $B^*/(\epsilon/2)$, and we want the number of extra bins to be $B = B^* + \frac{x}{k} \leq 1 + (1+\epsilon)B^* \implies k \geq \frac{x}{1+\epsilon B^*} = \frac{2B^*}{\epsilon(1+\epsilon B^*)} \geq \frac{2}{\epsilon^2}$). Since the rounding procedure generates a set of distinct sizes, we can use the algorithm in part (a).

- Then, take the remaining small items and place them into the bins. As per part (b), this will use at most $1 + (1+\epsilon)B^*$ bins.

Since each part of the algorithm is made up of polynomial steps, the overall runtime is also polynomial. Thus, we have given a polynomial time scheme that uses at most $1 + (1+\epsilon)B^*$ bins to pack the items.

Problem 5. (a) Consider the minimum spanning tree of G', which consists of edges with cost equal to the shortest path between terminals. Suppose we run a depth-first search on this MST starting from any arbitrary node, and for each edge (u, v) encountered during the DFS, we add the shortest path from u to v in G to the construction of a Steiner tree S. In some cases, the shortest path from u to v may involve some edges that have already been added to S already, in which case our construction of the Steiner tree will double those edges (whereas in OPT, they would only be added once). Thus, for each edge in the MST of G', we add a path with equivalent cost to the construction of S, which is at at most $2|OPT|$. Therefore, the total cost of the MST of G' (which is equal to the cost of S) is at most double that of the optimal Steiner tree.

(b) We can first pre-process the graph to compute all-pairs shortest paths, then create a complete graph G' on the terminals T with edge cost equal to the shortest path between edge endpoints. Then, we can generate the MST of this graph using Prim's or Kruskal's algorithm, and use the approach in part (a) to construct a Steiner tree S by processing the edges of the MST in depth-first search order. The constructed Steiner tree will have cost at most $2|OPT|$, giving us a 2-approximation algorithm.