Problem 1

Part (a)

I argue that the greedy algorithm (repeatedly take any edge that does not conflict with previous choices) can be implemented in linear time and gives a 2-approximation to the maximum (number of edges) bipartite matching.

The greedy algorithm can be implemented in linear time as follows. Keep 2 boolean arrays (one for each side of the matching) of size n indexed by the nodes on that side of the matching. The arrays indicate which nodes are still available, so initially all values are true. Process each of the m edge one after another: if the edge has both endpoint available (just check in the arrays), add the edge to the solution and mark its endpoints as unavailable in the arrays. This algorithm takes $O(m)$ since it has to process m edges and does a constant amount of work on each edge.

I show that this greedy algorithm gives a 2-approximation to the maximum bipartite matching. When the algorithm includes an edge in the solution, it makes at most 2 edges of the optimal solution unavailable (one for each endpoint). So if our algorithm ends with k edges, the optimal solution has $\leq 2k$ edges.

Part (b)

I generalize to argue that when edges have positive "weights", the greedy algorithm (consider edges in decreasing order of weight) can be implemented in $O(m \log n)$ time and is a 2-approximation algorithm for maximum (total) weight bipartite matching.

The linear algorithm can be implemented in $O(m \log n)$ time as follow. The procedure is similar to part (a), except that edges are processed in decreasing order of weight. So the algorithm must first sort the edges, which takes $O(m \log m)$ time, or actually $O(m \log n)$ time, since $m \leq n^2$. Thus, because of the sorting, the algorithm takes $O(m \log n)$ time overall.

I show that this greedy algorithm gives a 2-approximation to the maximum (total) weight bipartite matching. When the algorithm includes an edge of weight w in the solution, it makes at most 2 edges of the optiomal solution unavailable (one for each endpoint). Since these edges have not yet been processed, each has weight $\leq w$. So when our algorithm increases the solution by w, the optimal solution increases by $\leq 2w$. Therefore, if our algorithm ends with a total of k, the optimal solution has a total of $\leq 2k$.

Part (c)

I show that the same holds for the general (non-bipartite) max-weight matching problem.

The greedy algorithm from part (b) is modified to just have one array for nodes availability. It still runs in $O(m \log n)$ because the edges need to be sorted.

The argument to show that this greedy algorithm gives a 2-approximation to the general max-weight matching problem is similar to part (b). When the algorithm includes an edge of weight w in the solution, it makes at most 2 edges of the optimal solution, of weight $\leq w$ each, unavailable (one for each endpoint). So if our algorithms ends with a total of k, the optimal solution has a total of $\leq 2k$.

Problem 2

Part (a)

Supposing there are only k distinct item sizes for some constant k, I argue that I can solve bin-packing in polynomial time using dynamic programming.

My dynamic programming table has the n^k distinct job profiles (represented as a k-vector of integers from 0 to n) as one index variable and the n number of possible bins as the other. Let $B(p, j)$ be true if the profile p can fit into j bins and false otherwise.

I initialize $B(p, 1)$ as true if the profile p can fit into 1 bin and false otherwise. I then compute $B(p, j+1)$ recursively as whether p can be decomposed into two profiles, d and $p-d$, s.t. profile d can fit into 1 bin and profile $p-d$ can fit into j bins:

$$B(p, j+1) = \vee_{\forall d} \left(B(d, 1) \wedge B(p-d, j)\right)$$

To find the minimum number of bins for a particular profile p of items, simply search $B(p, j)$ for j from 1 to n for the first true entry and trace back to get the actual packing. Since the table has $n^k \cdot n$ entries and each entry can be computed in $O(n^k)$ time, the table can be filled in $O(n^{2k+1})$ time. So the algorithm is polynomial for a fixed k.

Part (b)

Supposing I have packed all items of size $> \epsilon$ into B bins, I argue that, in linear time, I can add the remaining small items to achieve a packing using at most $\max(B, 1 + (1+2\epsilon)B^*)$ bins.

I consider the B bins in some order, keeping a pointer to the current bin being considered. For each small item, I consider whether the item fits into the bin currently pointed at. If it doesn't, I increment the pointer and try the next bin, until I find a bin in which the item fits, creating a new bin if necessary. The algorithm is obviously linear in B plus the number of small items left to pack.

Suppose I end up with n bins. If $n = B$, then I achieve a packing using at most B bin. So suppose $n > B$. Notice that the $n-1$ first bins each have $< \epsilon$ space left. So the total space taken by all the items is $\geq n - 1 - (n-1)\epsilon = (n-1)(1-\epsilon)$. Now, the optimum number of bins, B^* must provide at least that much space. So $(n-1)(1-\epsilon) \leq B^*$. Hence, $n - 1 \leq \frac{B^*}{1-\epsilon} < (1+2\epsilon)B^*$ if $\epsilon \leq 1/2$. From which, I conclude that $n < 1 + (1+2\epsilon)B^*$ if $\epsilon \leq 1/2$. If $\epsilon > 1/2$, then $1 + (1+2\epsilon)B^* < 2B^*$, so I simply need to show that the average fill ratio of bins is $\geq 1/2$. Because $\epsilon > 1/2$, it is clear that the B bins are filled enough. Now, consider any two consecutive bins starting with the Bth bin. I had to create the second bin because I had an item that was overflowing the first bin, so the sum of items in the two bins is ≥ 1, thus the average fill of these two bins is $\geq 1/2$.

It follows that I can achieve a packing using at most $\max(B, 1 + (1+2\epsilon)B^*)$ bins.

Part (c)

Reducing to the previous case by rounding each item size up to the next power of $(1+\epsilon)$ doesn't work for bin packing because the bins have fixed size, so it's possible to double the number of bins required even by small increases in the sizes of the items (for example, if the items are close to 1/2 in size).

Part (d)

Notice that each item in S_j in its original size takes more space than each item in S_{j+1} in its increased size. So consider the optimal original packing and set aside the items in S_1. I can place the items in S_2 with their increased size in the space left by the items in S_1, ..., the items in S_j with their increased size in the space left by the items in S_{j-1}, ..., so that I can pack all items except those in S_1 with their increased sizes in the optimum original packing. Now, there are n/k items in S_1, and, in the worse case, I pack them in one bin each, so I increase the optimal number by at most n/k.

Part (e)

Let w be the number of items of size $> \frac{\epsilon}{2}$. Let B_w be the optimum number of bins for packing these w items. Using parts (a) and (d), I can pack these items in $B \leq B_w + \frac{w}{k}$. Setting $k = \frac{2}{\epsilon^2}$, $B \leq B_w + w\frac{\epsilon^2}{2}$. Notice that the space taken by the w items is $\geq w\frac{\epsilon}{2}$, so $w\frac{\epsilon}{2} \leq B_w$. Thus, $B \leq (1+\epsilon)B_w \leq (1+\epsilon)B^*$. So, using part (b) to add the remaining small items of size $\leq \epsilon/2$, I can conclude that I use at most $(1+2\epsilon)B^* + 1$ bins. By parts (a) and (b), this algorithm is polynomial for a fixed ϵ.

Problem 3

Part (a)

I argue that any feasible subset of jobs might as well be scheduled in order of increasing deadlines. Indeed, swapping adjacent out-of-order jobs doesn't change feasibility. Suppose that job i is scheduled just before job j but that $d_i > d_j$. Let t be the time that job i starts. Since the schedule is feasible, $t + p_i < d_i$ and $t + p_i + p_j < d_j < d_i$. If job i and job j are swapped, the schedule is still feasible because job j completes at time $t + p_j < d_j$ and job i completes at time $t + p_i + p_j < d_i$.

Part (b)

Assuming the lateness penalties are polynomially bounded integers, I give a polynomial-time dynamic program that finds the fastest-completing maximum-weight feasible subset. I re-index the jobs so that $d_1 \leq d_2 \leq \ldots \leq d_n$.

I define $B(j, w)$ to be the amount of time taken by the fastest-completing feasible subset of jobs $\in \{1, \ldots, j\}$ with weight $\geq w$.

I initiliaze $B(\emptyset, w)$ as follows:

$$B(0, w) = \begin{Bmatrix} 0 & \text{if } w \leq 0 \\ \infty & \text{otherwise} \end{Bmatrix}$$

I compute $B(j + 1, w)$ recursively from $B(j, w)$ as follows:

$$B(j + 1, w) = \min\left(B(j, w), \begin{Bmatrix} R(j, w) & \text{if } R(j, w) \leq d_{j+1} \\ \infty & \text{otherwise} \end{Bmatrix}\right)$$

where $R(j, w)$ is defined as:

$$R(j, w) = p_{j+1} + B(j, w - w_{j+1})$$

Let $W = \max_j w_j$. I search the table from $w = nW$ down for the first $B(n, w) < \infty$, which gives me the time $B(n, w)$ and the weight w of the fastest-completing maximum-weight feasible subset. By tracing back, I can explicitly get the feasible subset itself.

This dynamic programming algorithm takes time $O(n^2 W)$.

Part (c)

I use rounding and scaling to give a fully polynomial-time approximation scheme for the original problem of minimizing lateness penalty with arbitary lateness penalties.

Suppose the maximum weight of optiomal solution is Q. I round and scale each lateness penalty w_j:

$$w_j \rightarrow \left\lfloor \frac{n}{\epsilon Q} w_j \right\rfloor$$

Assuming $p_j \leq d_j \quad \forall j$, each scaled and rounded $w_j < \frac{n}{\epsilon}$. So I can run the dynamic programming algorithm in $O(\frac{n^3}{\epsilon})$ time. The weight of the optimal solution from the DP algorithm is just a scaled down Q with a possible rounding error for each j, so the optimum weight is $\geq \frac{n}{\epsilon Q}Q - n = n(\frac{1}{\epsilon} - 1)$. Scaling back up yields an optimum weight of $\geq n(\frac{1}{\epsilon} - 1)\frac{\epsilon Q}{n} = (1 - \epsilon)Q$. In order to hone in the optimal solution Q which allows a good scaling factor, I perform a multiplicative binary search between W to nW.

Thus, since my optimum is $\geq (1 - \epsilon)$ the real optimum, I have a polynomial-time approximation scheme.

Problem 4

Part (a)

I show that a minimum cycle cover can be found in polynomial time by reducing the problem to maximum weight bipartite matching.

Place the n vertices on one side of the bipartite matching as "entry" nodes and on the other side as "exit" nodes. For each edge of length l between v and w in the original graph, add an edge of weight $-l$ from the "entry" node of v to the "exit" node of w in the bipartite matching graph. Now, look for a maximum weight perfect matching in the bipartite matching graph. If the matching is perfect, this corresponds to a minimum cycle cover, where the matching edges represent edges that are in the cover.

Part (b)

Each cycle has at least 2 nodes, so there are $\leq n/2$ cycles. Therefore, there are $\leq n/2$ representative nodes. The optimum tour traversing only these representative nodes costs not more than the original optimum by the triangle inequality. Indeed, in the worse case, I can always just use the original optimum, traversing all nodes instead of just the representative ones.

Part (c)

I can unravel all the selections I've done, patching together various cycles to produce a tour of the whole graph. At the top-level, starting from the last iteration, I use the cycle cover found, stopping at each representative node, and opening up and tracing the cycle behind the representative node recursively. So I'll be visiting at least all nodes underneath this representative node before moving on to the next representative node, and so on, recursively.

Let C be the cost of the optimum tour. By part (b), I won't be paying more than C at each iteration. So $C(n) \leq C + C(\frac{n}{2})$. Thus, $C(n) = C \cdot O(\log n)$, so the tour of the whole graph costs $O(\log n)$ time the optimum.

Problem 5

Part (a)

I argue that the optimum strategy is to work on whichever job is available with the shortest remaining processing time.

Consider available jobs j and k with remaining processing times l_j and l_k. Let t be the current time. If I process job j then job k, I get $C_j = t + l_j$ and $C_k = t + l_j + l_k$ for a total $C_j + C_j = t + 2l_j + l_k$. Similarly, if I process job k then j, I get $C_j + C_k = t + 2l_k + l_j$. So if $l_j < l_k$, I am better off processing j first, and if $l_k < l_j$, I am better off processing k first.

Therefore, in general, at any point in time, I am better off processing the job with the shortest remaining processing time.

Part (b)

I show that scheduling the jobs non-preemptively in order of their completion time in the pre-emptive schedule increases each completion time only by a factor of 2.

Let C_j be the completion time of job j in the preemptive schedule. Let C'_j be the completion time of job j under the new non-preemptive schedule. I show that $C'_j \leq 2C_j$ by induction on the order of the job j under C_j.

base case: Consider the job j with the smallest C_j. It is the first job to complete under the pre-emptive schedule, so it cannot have been pre-empted (since then, it would need to wait until a job with a smaller remaining processing time completes – but there's no such job!). Therefore, $C_j = r_j + p_j$. In the non-preemptive schedule, job j is the first to run, so $C'_j = r_j + p_j$. Since, for the first job, $C_j = C'_j$, $C'_j \leq 2C_j$ trivially.

inductive case: Consider the job j with the $(i + 1)$th smallest C_j and the previous job p with the ith smallest C_j. Note that $C'_j = \max(r_j, C'_p) + p_j$. By the inductive hypothesis, $C'_p \leq 2C_p$, so $C'_j \leq \max(r_j, 2C_p) + p_j$. I consider two cases:

1. Under the pre-emptive schedule, job j ran uninterrupted from r_j to completion. So $C_p \leq r_j$ and $C_j = r_j + p_j$. Then, $C'_j = \max(r_j, 2C_p) + p_j \leq 2r_j + p_j \leq 2(r_j + p_j) \leq 2C_j$.
2. Under the pre-emptive schedule, job j completed (at least part of) its time after C_p. Decompose $p_j = b_j + a_j$ where b_j is the processing time completed before C_p and a_j is the processing time completed after C_p. Also, decompose $C_p = o_p + b_j$. So $r_j \leq C_p$ and $C_j = C_p + a_j = o_p + b_j + a_j$. I consider two cases:
   (a) If $b_j = 0$, then $C'_j \leq 2C_p + p_j = 2o_p + a_j \leq 2(o_p + a_j) \leq 2C_j$.
   (b) If $b_j > 0$, then notice that actually $C'_p \leq 2C_p - b_j$ as by C'_p j won't be processed at all. So $C'_j \leq 2C_p - b_j + p_j = 2o_p + 2b_j + a_j \leq 2(o_p + b_j + a_j) \leq 2C_j$.

Part (c)

I argue that the non-preemptive scheduling of part (b) yields a 2-approximation to (non-preemptive) average-completion-time scheduling with release dates.

Pre-emptive scheduling of part (a) is more relaxed than non-preemptive scheduling, and we achieve optimality in part (a). Therefore, the optimal preemptive scheduling represents a lower bound of what we can achieve in non-preemptive scheduling. Since the non-preemptive scheduling of part(b) yields a 2-approximation to the optiomal preemptive scheduling of part (a), it certainly also yields a 2-approximation to non-preemptive average-completion-time scheduling with release dates.

Problem 6

Part (a)

Here is the ILP to solve the problem $1 \mid prec \mid \sum w_j C_j$. The variable x_{jt} denotes the indicator that job j completed at time t exactly. For convenience, I introduce variables s_{jt}, derivable from x_{jt}'s, denoting the indicator that job j completed *before* or at time t.

$$\min \sum_j \sum_t w_j t x_{jt}$$

$$\begin{aligned}
&\text{subject to the constraints} \\
0 \leq x_{jt} &\leq 1 \quad \forall j, t \\
s_{jt} &= \sum_{t'=0}^{t} x_{jt'} \quad \forall j, t \\
\sum_t x_{jt} &= 1 \quad \forall j \\
s_{j(t+p_j)} &\leq s_{it} \quad \forall t, j, i \in A(j) \\
\sum_j p_j s_{jt} &\leq t \quad \forall t
\end{aligned}$$

Part (b)

For a given h_j, the smallest $\overline{C}_j$ happens when $x_{j0} = \frac{1}{2} - \epsilon$ and $x_{jh_j} = \frac{1}{2} + \epsilon$ as $\epsilon \to 0$. In this case, we have $\overline{C}_j = \frac{h_j}{2}$. So $\overline{C}_j \geq \frac{h_j}{2}$.

Part (c)

The halfway point h_j is the time t, s.t. $s_{jt} \geq \frac{1}{2}$ and $s_{j(t-1)} < \frac{1}{2}$. The precedence constraints ensure that $s_{j(t+p_j)} \leq s_{it} \quad \forall t, j, i \in A(j)$. Supposing $p_j \geq 1$, the precedence constraints at least ensure that $s_{j(t+1)} \leq s_{it} \quad \forall t, j, i \in A(j)$. So $\forall j, i \in A(j)$, at the time $h_i - 1$, where $s_{i(h_i-1)} < \frac{1}{2}$, the precedence constraints ensure that $s_{jh_i} < \frac{1}{2}$, which means that $h_j > h_i$. Thus, no job runs before its predecessors.

Part (d)

Let $P(j)$ comprise job j and all the jobs i such that job i precedes job j in the given order. At h_j, at least half of each job in $P(j)$ has been completed, so $\frac{1}{2}\sum_{i \in P(j)} p_i \leq h_j$. Hence, $\sum_{i \in P(j)} p_i \leq 2h_j \leq 4\overline{C}_j$ (by part (b)). Thus, the actual completion time for job j is at most $4\overline{C}_j$.

Part (e)

The LP is more relaxed than its ILP. By part (d), we achieve a 4-approximation to the LP, so we certainly achieve a 4-approximation to the ILP. Thus, we have a constant-factor approximation to $1 \mid prec \mid \sum w_j C_j$.