$\frac{10}{10}$

1. A. We'll use the method described in class in the proof of optimality criteria 2. We'll add a new vertex s with outgoing edges to all other vertices with cost 0 and calculate the minimum distance from s to each of the other vertices. These distances are well defined since we know there are no negative cycles (according to opt. criteria 1). We set $\pi(v) = \delta(s, v)$.
Proof: Assume by contradiction for some edge (v,w), $c^{\pi}((v, w)) = cost(v, w) + \pi(v) - \pi(w) < 0$. That means that $\pi(w) = \delta(s, w) > \pi(v) + cost(v, w) = \delta(s, v) + cost(v, w)$. This means that we can find a path from s to w using (v,w) at a cost lower than $\delta(s, w)$ – a contradiction.
Running time: adding vertex s with n edges - O(n).
Finding single-source shortest paths can be done with bellman ford in $O(mn)$.
Calculating the potentials can be done in $O(1)$ per node.
In total – $O(mn)$.

B. We'll start with the potentials given to us and the reduced costs they induce. First we'll saturate all edges with negative reduced-cost. Afterwards we will ignore edges with positive reduced cost and use only zero-reduced cost edges to get rid of excesses and deficits using the max flow algorithm we've seen in class in the following way:
We'll add 2 nodes, s and t, and create edges between s and every node with excess with capacity equal to its excess. Similarly, we'll create an edge between each node with a deficit to t, with capacity equals to its excess. Next, we'll run the max-flow algorithm from class using only the new edges and the edges with 0 reduce cost. The minimum cost circulation is the one formed without the edges involving s and t.
Correctness: First we need to prove we indeed have a circulation, for that we need to show that all edges from s and t are saturated. To prove this we need to show that the max flow is equal to the sum of all excesses – E. The max flow isn't greater than E, because there's a cut {s}\{all other nodes} that is of that size. The max flow can't be lower than E because that would indicate we have an edge with a negative reduced cost with every circulation, contradicting π's optimality. Therefore we have a legal circulation, without any negative reduced cost arcs. According to optimality criteria 2, this is a min-cost max flow.
Complexity: saturating all negative reduced costs can be done in $O(m)$, looking only at zero reduced cost edges can be done at $O(m)$, adding edges from source and to sink can be done in $O(n)$, and the max-flow algorithm we've seen takes $O(mnlog\left(\frac{n^2}{m}\right))$. In total the algorithm takes $O(mnlog(\frac{n^2}{m}))$ time to run.

$\frac{10}{10}$

2. A. First we'll define a potential function $\pi(v) := \textit{length of shortest path from } s \textit{ to } v$. We'll prove by induction that there never will be a negative reduced-cost cycle in the residual graph by proving that there will never be a negative reduced-cost edge in the residual graph.

Proof: It holds in the beginning of the algorithm, because by the definition of π, if (v,w) is on a shortest path then $\pi(v) + cost((v,w)) - \pi(w) = 0$, otherwise it's greater than 0. After finding a shortest path from s to t, each edge (v,w) on the path has a reduced cost 0. When we push flow through this shortest path, the new residual backwards arcs are of reduced-cost -0=0, therefore keeping all reduced-cost edges non-negative.

This holds throughout the algorithm, and in particular at the end of the algorithm. There are no negative cycles, and there is no path from s to t in the reduced graph, therefore when reducing to the min cost circulation problem we'll have no negative cycles.

B. We can implement a min-cost flow algorithm using minimum cost augmenting paths in the following way: First we'll run Dijkstra once from s, finding shortest paths from s. We'll then define $\pi(v) := \textit{length of shortest path from } s \textit{ to } v$. From now on, we'll only look at the reduced-costs induced by the potential function. On each step we'll find shortest reduced-cost path from s-t and push maximum possible flow on that path (that is the minimum edge capacity on the path) and then run Dijkstra again to find shortest paths and calculate the new potential function. When we are done, according to the proof in (A) we have a min-cost flow (we can run Dijkstra because all reduced costs are always non-negative).

You assume here that the graph is simple. In general, it would be $O(mU)$

Complexity analysis: we need to run Dijkstra once in the beginning, and then Dijkstra in each step. Each step increases the flow by at least 1, because all capacities are integers, and the minimum capacity for a reduced edge is 1. The maximum possible flow is at most (n-1)U, because there are at most (n-1) edges coming out of the source, each with at most U capacity.

Therefore the algorithm halts after at most nU steps, each of those steps cost $O(m + nlogn)$, and in total $O(mnU + n^2Ulogn)$.

10/10

3. A. In order to find a minimum cost flow, we'll first add 2 nodes to graph, s and t. We'll add outgoing edges from s to all nodes in V_1 and outgoing edges from all nodes in V_2 to t, of cost 0 (the cost doesn't really matter).
We'll set the capacity of each edge in the new graph to 1, and run the min-cost flow algorithm from class (actually we've seen a min-cost circulation algorithm in class, but we've seen how to reduce the min-cost flow to the min-cost circulation in class by adding an outgoing edge from t to s).
Correctness: First it's easy to see the maximum flow is n. We can find a flow of n simply by saturating all the paths of the following kind: $s \to V_{1,i} \to V_{2,i} \to t$, and there's a cut of size n (for example {s} and all the rest).

Second, we know that by using the algorithm from class, we'll always push flow of integer amount on each edge. This is true because the push operation always pushes $\min(e(v), r(v,w))$ which is an integer (easy to prove by induction).

As a result, the algorithm will saturate each edge from $s \to V_1$, and each edge from $V_2 \to t$ (this is the only way the flow will be n). This means that only one edge from each node in V_1 will have flow pushed on it (because the flow is integer – either 0 or 1, and the total flow is 1). Similarly, each node in V_2 will have one incoming edge with flow on it (because it has an incoming flow of 1). This shows we have a perfect match. Because each perfect match is isomorphic to a max flow, the min-cost max flow, we'll be isomorphic to the min-cost perfect match.

B. We'll create a node v_i for the ith location and 2 nodes, s and t. We'll create the following edges of cost 0 and capacity 50: (s, v_1), $\forall 1 \le i < 50.\ (v_i, v_{i+1})$, (v_{50}, t).

We also create the point-to-point edges, for each $i < j$, create an edge (i, j) of cost $-f_{i,j}$ and capacity $q_{i,j}$ (assuming $q_{i,j}$ is at most 50, otherwise we'll set capacity to 50).

Next we run the min-cost algorithm. According to the incoming and outgoing flow to each node, the bus driver can tell how many people to load/drop. The incoming flow from point-to-point edges, signals the amount of passengers to drop, and the amount of the outgoing point-to-point edges signals the amount of people on the bus when leaving the station.

Correctness: It's easy to see the bus will never have more than 50 people on it, because the capacity of all edges is at most 50. Each flow on the point-to-point edges is actually the amount of people using the bus to travel between the 2 points and therefore the min-cost flow matches the maximum profit (because we negated the prices) the bus driver can make.

edge might be contained in all [illegible] F_i's

The counters shall be per component, and not per level

$\frac{10}{10}$

5. A. We'll use a 1d-segment tree to hold all the segments as seen in class, but instead of saving with each node v all the intervals that contain Interval(v) and not Interval(parent(v)), we'll only keep the one with the largest cost. The query will go over a path in the segment tree and find the interval with the largest cost. The query takes $O(logn)$ time because the length of the path is $O(logn)$. The construction is similar to the regular segment tree construction, but instead of adding intervals to the subset of each node, we'll only keep the largest cost one so far. Because we only save one interval with each node, the size of the data structure is proportional to the number of nodes in the tree which is $O(n)$.

B. We'll use dynamic trees to store the forest as described in the question, i.e. s_1 is the parent of s_2 if s_1 contains s_2 and there is no segment s that is contained in s_1 and contains s_2. To make things simpler, we'll add an interval $(-\infty, \infty)$, with minimal cost, in order to turn the forest into a single tree. Given a point query p, we need to find the smallest interval I containing p, and return the highest cost interval on the path between I and the root of the tree. This can be done using a maxcost dynamic tree and using the maxcost operation (we can also negate all the costs and use a regular mincost dynamic tree). In order to find the smallest interval I containing p, we'll hold an auxiliary binary search tree, on the end points of the intervals, and we'll hold pointers between each end point and its corresponding interval in the dynamic tree. Using this tree we can find the interval with the closest end point to p in the dynamic tree. That interval is either I or one of its children.

Delete and insertion operation might take a long time with regular dynamic trees, because the degree of each node can be high, and therefore the number of cuts and links we'll have to do is also high (for example if we have a node v, with n children, $v_1, \dots v_n$, and we add a new interval contained in v and containing $v_1, \dots v_n$ we'll need to do $O(n)$ cuts and links to fix the dynamic tree). In order to avoid this situation, we'll keep the degree of nodes in the dynamic tree low by using the method we've seen in class. We'll replace each node v with a path the length of v's degree. Each node in the path will actually hold only one interval which was a child of v in the regular dynamic tree. This will make the delete and insert operation quicker, but will increase the number of nodes in the tree to at most $O(n^2)$ (note that it doesn't change the asymptotic running time of the dynamic tree, since $O(logn) = O(\log(n^2))$.

The query operation will first find the interval I in the secondary search tree, and use the dynamic tree to calculate the max-cost on the path to the root. If I is the smallest interval containing p, then the algorithm will return $\max(maxcost(I), cost(I))$, otherwise I is a child of the smallest interval and we can simply return $maxcost(I)$. The query takes $O(logn)$ to find I in the secondary tree + $O(logn)$ amortized to calculate $maxcost$ - $O(logn)$ amortized in total.

Insert operation: we'll update the secondary search tree with the endpoints of the interval and add pointers from them to the new node in the dynamic tree. We can tell where we need to insert the new interval $[i_1, i_2]$ by using a query on the 4 points next to i_1 and i_2. We can then use a constant number of links and cuts to insert the new interval into the tree. Finding and adding the interval to the secondary search tree takes $O(logn)$ time. Finding the place and adding the interval to the dynamic tree takes $O(logn) + O(logn)$ amortized time. In total the insertion operation takes $O(logn)$ amortized time.

Delete operation: We'll find and delete the interval I from the secondary search tree. We'll then find the interval in the dynamic tree, and remote it from the dynamic tree using a constant number of links and cuts (linking $I's$ children to its parent, and cutting I from its parent). The running time of finding and deleting the interval from the secondary tree is $O(logn)$, and finding and deleting the interval from the dynamic tree takes $O(logn) + O(logn)\ amortized.$ In total the running time of delete operation is $O(logn)$ amortized time.