Problem Set 3 Solutions

Calvin Walker

Problem 1:

Algorithm: Take the schedule S_{greedy} such that $(i_1, \ldots, i_n)$ are in decreacing order of c_i, i.e. each day, return the book to the library with the highest remaining late fee.

Runtime: This only requires sorting the libraries in decreacing order of c_i, so the runtime is $O(n \log n)$.

Correctness: Given an optimal schedule S such that $c_{i_j} < c_{i_{j+1}}$, consider swapping i_j and i_{j+1} to obtain a new schedule S'. Observe that the cost incurred by the late fees for all $j' \in [n] \setminus \{j, j+1\}$ are unaffected, since the cost incurred by book j', $(j'-1)\ c_{i_{j'}}$ is unchanged. Since $c_{i_j} < c_{i_{j+1}}$,

$$-c_{i_{j+1}} < -c_{i_j} \tag{1}$$

$$j \cdot (c_{i_{j+1}} - c_{i_j}) - c_{i_{j+1}} < -c_{i_j} + j \cdot (c_{ij+1} + c_{i_j}) \tag{2}$$

$$(j-1)c_{i_{j+1}} + jc_{i_j} < (j-1)c_{i_j} + jc_{i_{j+1}} \tag{3}$$

So $c_{i_{j+1}}$ and c_{i_j} incur less late fees in schedule S' than in schedule S, so the late fees incurred by schedule S' must be less than or equal to schedule S. We can repeat this process until we obtain S_{greedy}, the schedule sorted in descending order of c_i, which must also be optimal.

Problem 2:

a) Algorithm: Starting with the largest denomination $d_1 = 10$, use $\lfloor \frac{N_j}{d_j} \rfloor$ bills of denomination d_j, set the remaining total to $N_{j+1} = N_j \mod d_j$, and set d_{j+1} to the highest unused denominatinon. Terminate when $N_j = 0$.

b) Let $S_{greedy} = (i_1, i_2, i_3, i_4)$ be the number used of each denomination d_i given by the greedy algorithm such that the denominatinon are in decreacing order. Consider an alternative allocation $S' = (i'_1, i'_2, i'_3, i'_4)$, that is also optimal. Let k be the first denomination in which the allocations S_{greedy} and S' differ. Since the greedy algorithim always takes the greatest amount of each denomination at each step, it must be that $i'_k < i_k$. Consider the following cases:

- k = 1: If $i'_1 < i_1$, then there must be some combination of i'_2, i'_3, i'_4 that sum to $d_1(i_1 - i'_1)$, which is a factor of 10, so these can be exchanged for an additional 10 dollar bill, and we use less bills than before, obtaining a partial solution $S = (i_1)$
- k = 2: Similarly, there must be some combination of i'_3, i'_4 that sum to $d_2(i_2 - i'_2)$, which is a factor of 5, so these can be exchanged for an additional 5 dollar bill, and we use less bills than before, obtaining a partial solution $S = (i_1, i_2)$
- k = 3: Again, there must enough i'_4 to make up the difference $d_3(i_3 - i'_3)$, which is a factor of 2, so these can be exchanged for an additional 2 dollar bill, and we use less bills than before, obtaining a partial solution $S = (i_1, i_2, i_3)$

Observe that in each case we exchange for less total bills, and can take the partial solution and continue the process for the next k. Since i_1, i_2, i_3 become fixed, i_4 is already determined, as there remains only one way to sum up to N. So the optimal solution $S = (i_1, i_2, i_3, i_4)$ is the same as S_{greedy}.

c) No the greedy algorithim would not always be optimal. Consider the following counter example: $N = 15$, so the greedy algorithim yeilds one 8 dollar bill, one 5 dollar bill, and three 1 dollar bills. But the optimal solution is three 5 dollar bills.

Problem 3:

a) Proof by induction on i, the current phase of the algorithm. For the base case $i = 1$, each *blue tree* is a single vertex with no edges, so every *blue tree* is a tree. For the inductive step assume that at phase k every *blue tree* is a tree. During phase $k+1$ each blue tree only obtains new verticies via new edges, so they must remain connected. Every *blue tree* gains an edge e that crosses a cut between two *blue trees*, so e can never create a cycle. Therefore, the *blue trees* are connected and acyclic throughout the algorithm, so they are always trees.

b) Proof by induction on i, the current phase of the algorithm. For the base case $i = 1$, each *blue tree* is a single vertex with no edges, so each *blue tree* must be a component of the mimimal spanning tree of G. For the inductive step assume that at phase k every *blue tree* is a component of the minimum spanning tree of G. During phase $k+1$, we define a cut $C = (L, R)$ between each *blue tree* and the remaining other *blue trees*. The algorithm chooses the shortest edge crossing C, which by theorem 5.6 must be in the minimum spanning tree of G. So at the end of phase $k+1$ all edges in the *blue trees* must be in the minimum spanning tree of G, and the inductive case holds.

c) We start the procedure with n *blue trees*, which in the second phase each connect to at least one other *blue tree*, so the number of *blue trees* at least halves at each step. So there are at most $\frac{n}{2^{t-1}}$ *blue trees* at the end of phase t. The MST is found when there is only one *blue tree* remaining, ie. $\frac{n}{2^{t-1}} = 1$, so there are at most $t = \log_2(n) + 1$ phases. This bound is tight for the following adjacency list:

$$
\begin{aligned}
u_1 &: (1, u_1') \\
u_1' &: (1, u_1), (n+1, u_2') \\
u_2 &: (1, u_2') \\
u_2' &: (1, u_2), (n+1, u_1'), (n+2, u_3') \\
&\vdots \\
u_n &: (1, u_n') \\
u_n' &: (1, u_n), (n+(n-1), u_{n-1}')
\end{aligned}
$$

Since a single vertex could be the the closest vertex to the $n-1$ other verticies in G, we can contruct a graph for any n that terminates in exactly two phases via the following adjacency list:

$$
\begin{aligned}
v &: (1, u_1), (2, u_2), \ldots, (n, u_n) \\
u_1 &: (1, v) \\
u_2 &: (2, v) \\
&\vdots \\
u_n &: (n, v)
\end{aligned}
$$

So the number of phases t until termination is $2 \leq t \leq \log_2(n) + 1$

Problem 4:

Algorithm: Let a_k be the item at the k'th index of A. Initialize variable $i = 0$ and variable j to the last index of the array. At each iteration, take $k = \lfloor \frac{i+j}{2} \rfloor$, and index into find the value a_k

(a) If $a_k = k + 1$, repeat with $i' = k + 1$ and $j' = j$

(b) If $a_k < k + 1$, repeat with $i' = i$ and $j' = k - 1$

Once $i = j$ the process terminates, and the target index is equal to $i+1$ or i, which can be found in constant time by comparing $A[i-1], A[i]$. If $A[i-1] = A[i]$, return i. Otherwise return $i + 1$.

The algorithm is quite intuitive. If there is a duplicate number prior to the current index k, then $a_k < k+1$, so we can reduce the problem to the portion of the A preceding k. If $a_k = k + 1$, then no element has been repeated, since the index properly aligns with the integer, so the problem can be reduced to the half A after index k. This repeats until there is only one possible candidate, which must either be the first or second instace of the duplicate.

The algorithim modifies the binary search algorithim to find the correct index, and shares its recuurence relation $T(n) = T(\frac{n}{2})+1$ for a runtime of runtime of $O(n \log n)$.