**CS161: Design and Analysis of Algorithms** **Summer 2004**

# Problem Set #5 Solutions

## General Notes

- **Regrade Policy:** If you believe an error has been made in the grading of your problem set, you may resubmit it for a regrade. If the error consists of more than an error in addition of points, please include with your problem set a detailed explanation of which problems you think you deserve more points on and why. We reserve the right to regrade your entire problem set, so your final grade may either increase or decrease.

1. **[25 points]** Coin Changing

   Consider a currency system with $k \geq 1$ distinct (integer) denominations $d_1 < \ldots < d_k$. Assume that $d_1 = 1$ so that any integer amount greater than 0 can be formed. Throughout the problem, assume there are always coins of every denomination available when needed (i.e. you cannot "run out" of any denomination). Our goal will be to determine how to make change using the fewest number of coins.

   Some parts of this problem are similar to problem 16-1 on page 402 of CLRS. You may wish to read the statement of that problem to see how similar questions are presented.

   (a) **[2 points]** Describe a greedy strategy for making change using as few coins as possible. This strategy should work for the US coin system which uses denominations $\{1, 5, 10, 25\}$. You do not need to prove the correctness of this strategy for this set of denominations.

   **Answer:** The greedy strategy consists of always taking the largest denomination that we can at the time. We repeat this until we have the correct amount of change.

   MAKE-CHANGE($M$)
   1 **for** $i = k$ **downto** 1
   2 $\quad count[i] \leftarrow \lfloor M/d_i \rfloor$
   3 $\quad M \leftarrow M - count[i] \cdot d_i$

   Since for each denomination we calculate the number of coins we can take until we would make more than the amount of change asked for, this operation takes time $O(k)$.

   We can also write this in a slightly less efficient manner that will be useful when analyzing the correctness of the greedy strategy. In this case, we simply take one coin at a time of the largest denomination possible.

   MAKE-CHANGE-SLOW($M$)
   1 **for** $i = k$ **downto** 1
   2 $\quad$ **while** $M \geq d_i$
   3 $\quad\quad count[i] \leftarrow count[i] + 1$
   4 $\quad\quad M \leftarrow M - d_i$

   This runs in time $O(C + k)$ where $C$ is the number of coins in the greedy solution.

(b) **[5 points]** Suppose that $d_i = c^{i-1}$ for some integer $c > 1$, i.e. the denominations are $1, c, c^2, \ldots, c^{k-1}$. Prove that the greedy strategy from part (a) is optimal.

**Answer:** We'll represent a change solution by a series of counts, one for each coin denomination. The counts must obey the property $\sum_{i=1}^{k} count[i] \cdot d_i = M$. The number of coins in this solution is $\sum_{i=1}^{k} count[i]$. Note that the counts for each coin denomination $d_i < d_k$ must be strictly less than $c$ in an optimal solution. Otherwise, if we have $count[i] \geq c$, we can replace $c$ of the $d_i = c^{i-1}$ coins which have value $c * c^{i-1} = c^i$ with one $d_{i+1} = c^i$ coin and get a smaller count.

We will prove by contradiction that the greedy solution is optimal. Let the optimal count values be represented by $optimal[i]$. Let $j$ be the first index (highest) for which $count[j] \neq optimal[j]$. We know that $count[j] > optimal[j]$ because the greedy solution always takes the maximum number of any coin and this is the first value on which they disagree. If the optimal solution chose more of coin $d_j$, then it would have made too much change. Let $B = \sum_{i=1}^{j-1} count[i] \cdot d_i$, the amount of change that the greedy solution makes with the coins $1 \ldots j-1$. Since the optimal solution used fewer coins of denomination $d_j$, we know that the amount of change that the optimal solution makes with coins $1 \ldots j-1$ is $D = B + (count[j] - optimal[j]) \cdot d_j = B + (count[j] - optimal[j]) \cdot c^{j-1}$. Since this value is greater than $c^{j-1}$, we will show that the optimal solution must take at least $c$ coins for some coin $d_1 \ldots d_{j-1}$.

Assume that the optimal solution has $optimal[i] < c$ for $i = 1 \ldots j-1$. Then, the amount of change that the optimal solution makes with the first $j-1$ coins is

$$\begin{aligned} D &= \sum_{i=1}^{j-1} optimal[i] \cdot d_i \\ &\leq \sum_{i=1}^{j-1} (c-1) \cdot c^{i-1} \\ &= (c-1) \sum_{i=1}^{j-1} c^{i-1} \\ &= (c-1) \sum_{i=0}^{j-2} c^i \\ &= (c-1) \frac{c^{j-1} - 1}{c-1} \\ &= c^{j-1} - 1 \end{aligned}$$

However, this contradicts the fact that $D$ was greater than $c^{j-1}$. Thus, the optimal solution must take at least $c$ coins for some denomination less than $d_j$. But, as we showed above, this solution can't be optimal. This is a contradiction and therefore the greedy solution must take the same number of coins for all coins as the optimal solution, and thus must itself be optimal.

(c) **[2 points]** Give a set of coin denominations for which the greedy algorithm will

not always yield the optimal solution. Your set should include $d_1 = 1$ so that every positive integer amount can be formed.

**Answer:** A simple example is the set of US denominations without the nickel. If you simply have the values $\{1, 10, 25\}$, the greedy solution will fail. For example, if you are trying to make change for 30 cents, the greedy solution will first take a quarter, followed by 5 pennies, a total of six coins. However, the optimal solution actually consists of three dimes.

(d) **[5 points]** Suppose you wish to make change for the amount $M$ using an arbitrary set of denominations (although we still make the assumption that $d_i$ are distinct integers and $d_1 = 1$). Give an $O(Mk)$ time algorithm to calculate the minimum number of coins needed. State the (asymptotic) space usage of your algorithm in terms of the relevant parameters.

**Answer:** We can solve this problem using dynamic programming. Notice that for any value $M$, we can calculate the optimal solution by looking back at the number of coins needed to make $M - d_i$ coins for all $i$ and choosing the smallest number. We fill an array $T$ which ranges from $1 \ldots M$. At each point in the array, we store the optimal number of coins needed to make change. We initialize the cell $T[0]$ with the value 0. Then, for every other cell up to $T[M]$ we calculate

$$T[i] = 1 + \min_{1 \le j \le k} T[i - d_j]$$

where we assume that for $x < 0$, $T[x] = \infty$. We fill in the table up till $T[M]$ and then simply look in that cell for the optimal number of coins. Since we have to fill in $M$ values in the table and each value looks back at $k$ different cells and takes the minimum, the running time of this algorithm is $O(Mk)$. At any time, we can look back until the $i - d_k$ cell, so the space usage is $O(\min(d_k, M))$.

(e) **[5 points]** Suppose Eugene owes Nina an amount $M$. Give an $O(Mkd_k)$ time algorithm to determine the minimum number of coins that need to exchange hands in order to settle the debt (Nina can give change). Assume that neither Eugene nor Nina will run out of any denomination. State the (asymptotic) space usage of your algorithm in terms of the relevant parameters.

**Answer:** We can define an upper bound on the amount of money that Eugene gives Nina that can be part of an optimal solution. Let's say that Eugene gives Nina an amount more than $Md_k$ and Nina gives change. Eugene gives at least $M$ coins, and Nina still gives back coins. But, we know that Eugene could simply have given Nina $M$ coins of denomination $d_1 = 1$ to make a total value of $M$ without Nina having to give any change. So, we know that Eugene giving an amount more than $Md_k$ can't be an optimal solution.

We can therefore use our solution from part (d) to find the optimal ways of giving change for all values $1 \ldots Md_k$. Then, for each value $i$ that Eugene can give in the range $M \ldots Md_k$, Nina will need to give $i - M$ change. We simply sum the number of coins that Eugene gives plus the number of coins that Nina gives: $T[i] + T[i - M]$. The minimum over all $i$ is the optimal way to settle the debt.

To find the optimal ways to make change for all values up till $Md_k$ takes time $O(Mkd_k)$ using the algorithm in part (d). Notice that in the process of computing the optimum for $Md_k$ the algorithm finds the optimal ways to make change for all the values $1 \ldots Md_k$ so we only need to run it once. Summing and taking the minimum will take time $O(Md_k)$, so the running time is bounded by the dynamic programming part. The algorithm will need to store the previous $d_k$ solutions for the dynamic programming and the previous $M$ solutions for the summation of $T[i] + T[i - M]$. So, the space usage of the algorithm is $O(d_k + M)$.

(f) **[6 points]** Consider again the greedy algorithm stated in part (a) of the problem. We want to decide, in a given currency system $d_1 < \ldots < d_k$, whether the greedy algorithm is optimal. Give a dynamic programming algorithm to determine the smallest amount $M$ for which the greedy algorithm fails to produce the optimal solution. If the greedy approach is always optimal your algorithm should detect and report this. State the (asymptotic) running time and space usage of your algorithm in terms of the relevant parameters. You may assume that $k \geq 2$ so that the greedy algorithm is not trivial.

**Hint:** Can you find an upper bound $B$ such that if greedy is optimal for all $M \leq B$, then greedy must be optimal for all $M$?

**Answer:** We claim that if a counterexample to greedy's optimality exists, it must exist for some $M < d_k + d_{k-1}$. The main idea is that for such $M$, the greedy choice is to choose $d_k$, but we will show that doing so either yields an optimal solution or a smaller counterexample to greedy's optimality.

Thus, suppose (for a contradiction) that $M \geq d_k + d_{k-1}$ is the smallest amount such that greedy is not optimal for this amount. This means that $OPT[M] > OPT[M - d_k] + 1$. However, there must be some denomination $d_j \neq d_k$ such that $OPT[M] = OPT[M - d_j] + 1$ (recall how we computed $OPT[i]$ in part (d)). By assumption, greedy must be optimal for this amount, otherwise we have a smaller counterexample. However, since $d_j \leq d_{k-1}$ (since $j \neq k$), this means the optimal greedy choice for $M - d_j \geq d_k$ is in fact $d_k$; therefore, $OPT[M - d_j] = OPT[M - d_j - d_k] + 1$. Thus,

$$OPT[M] = OPT[M - d_j - d_k] + 2$$

However, notice that we can go from $M - d_k - d_j$ to $M - d_k$ using only the coin $d_j$; therefore, $OPT[M - d_k] \leq OPT[M - d_k - d_j] + 1$. Combining this with the first inequality we established yields

$$OPT[M] > OPT[M - d_k - d_j] + 2$$

for the desired contradiction.

Thus, we simply need to compute the optimal solutions for $M = 1 \ldots d_{k-1} + d_k$, using the algorithm from part (d). As we compute optimum, we must check that greedy is still optimal by checking whether $OPT[M] = OPT[M - d_j] + 1$, where $d_j$ is the largest denomination $\leq M$. If we fail to find a counterexample before $M = d_k + d_{k-1}$, we may stop and report that greedy is always optimal.

The space requirement $O(d_k)$ and the running time of $O(kd_k)$ are obtained by letting $M = O(d_k)$ in the results from part (d).

2. **[16 points]** Amortized Weight-Balanced Trees

   Throughout this problem, your explanations should be as complete as possible, but they do not need to be formal. In particular, rather than proving tree properties by induction you may simply argue why they should hold.

   (a) **[3 points]** Do problem 17-3(a) on page 427 of CLRS.
   **Answer:** We first perform an in-order walk starting from $x$ and store the sorted output in an array. This will require space $\Theta(size[x])$. Then, to rebuild the subtree rooted at $x$, we start by taking the median of the array which we can find in constant time (by calculating its address in the array) and put it at the root of the subtree. This guarantees that the root of the subtree is 1/2-balanced. We recursively repeat this for the two halves of the array to rebuild the subtrees. The total time for the algorithm follows the recurrence $T(size[x]) = 2T(size[x]/2) + 1 = \Theta(size[x])$.

   (b) **[2 points]** Do problem 17-3(b) on page 427 of CLRS.
   **Answer:** If we perform a search in an $n$-node $\alpha$-balanced binary search tree, the worst case split at any point in a subtree with $i$ nodes is recursing to a subtree with $\alpha i$ nodes since we know that the number of nodes in any subtree is bounded by this value. So, the recurrence is $T[n] = T[\alpha n] + 1 = \Theta(\log_{1/\alpha} n)$, which since $\alpha$ is a constant is simply $\Theta(\log n)$.

   (c) **[3 points]** Do problem 17-3(c) on page 427 of CLRS.
   **Answer:** If we define the potential of the tree as

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x)$$

   where $\Delta(x) = |size[left[x]] - size[right[x]]|$, then the potential for any BST is non-negative. This is easy to see as every term in the summation is the absolute value of the difference in the subtree sizes and therefore is non-negative. Also, we assume that the constant $c$ is positive, thus giving a positive potential $\Phi(T)$.

   For a 1/2-balanced BST, we know that $size[left[x]] \leq (1/2)size[x]$ and $size[right[x]] \leq (1/2)size[x]$ for all nodes $x$ in the tree. We will prove by contradiction that $\Delta(x) < 2$ for all $x$ in the tree. Assume that $\Delta(x) \geq 2$. Then, $|size[left[x]] - size[right[x]]| \geq 2$. Without loss of generality, assume that the left subtree is larger, so we can drop the absolute value signs. We also know that the sum of the sizes of the left subtree and the right subtree must be $size[x] - 1$. Substituting this in for $size[right[x]]$, we find

$$\begin{aligned} size[left[x]] - size[right[x]] &\geq 2 \\ size[left[x]] - (size[x] - size[left[x]] - 1) &\geq 2 \\ 2 \cdot size[left[x]] &\geq 1 + size[x] \\ size[left[x]] &\geq 1/2 + (1/2)size[x] \end{aligned}$$

But, we know from the $\alpha$-balanced condition that $size[left[x]] \leq (1/2)size[x]$, which is a contradiction. Therefore, for all nodes $x$, $\Delta(x) < 2$. The summation in the potential equation is only over nodes with $\Delta(x) \geq 2$. Since no such nodes exist, the summation is 0, and thus $\Phi(T) = 0$ for a 1/2-balanced tree.

(d) **[4 points]** Do problem 17-3(d) on page 427 of CLRS.

**Answer:** We need to pick the constant $c$ such that at any time that we need to rebuild a subtree of size $m$, we have that the potential $\Phi(T)$ is at least $m$. This is because the amortized cost of rebuilding the subtree is equal to the actual cost plus the difference in potential. If we want a constant amortized cost, we need

$$\begin{aligned} \hat{c}_{rebuild} &= c_{rebuild} + \Phi_i - \Phi_{i-1} \\ O(1) &= m + \Phi_i - \Phi_{i-1} \\ \Phi_{i-1} &\geq m \end{aligned}$$

since we know that the end potential $\Phi_i$ is always greater than zero.

We need to figure out the minimum possible potential in the tree that would cause us to rebuild a subtree of size $m$ rooted at $x$. $x$ must not be $\alpha$-balanced, or we wouldn't need to rebuild the subtree. Say the left subtree is larger. Then, to violate the $\alpha$-balanced criteria, we must have $size[left[x]] > \alpha \cdot m$. Since the size of the subtrees must equal $m-1$, we know that $size[right[x]] = m-1-size[left[x]] < m - 1 - \alpha \cdot m = (1-\alpha)m - 1$.

We have

$$\begin{aligned} \Delta(x) &= size[left[x]] - size[right[x]] \\ &> \alpha \cdot m - ((1-\alpha)m - 1) \\ &= (2\alpha - 1)m + 1 \end{aligned}$$

We can then bound the total tree potential $\Phi(T) > c((2\alpha-1)m+1)$. We need the potential larger than $m$ to pay the cost of rebuilding the subtree, so

$$\begin{aligned} m &\leq c((2\alpha-1)m+1) \\ c &\geq \frac{m}{(2\alpha-1)m+1} \\ &= \frac{1}{2\alpha - 1 + 1/m} \\ &\geq \frac{1}{2\alpha} \end{aligned}$$

Therefore, if we pick $c$ larger than this constant based on $\alpha$, we can rebuild the subtree of size $m$ in amortized cost $O(1)$.

(e) **[4 points]** Do problem 17-3(e) on page 427 of CLRS.

**Answer:** The amortized cost of the insert or delete operation in an $n$-node $\alpha$-balanced tree is the actual cost plus the difference in potential between the two states. From part (b), we showed that search took time $O(\lg n)$ in an $\alpha$-balanced tree, so the actual time to insert or delete will be $O(\lg n)$. When we insert or

delete a node $x$, we can only change the $\Delta(i)$ for nodes $i$ that are on the path from the node $x$ to the root. All other $\Delta(i)$ will remain the same since we don't change their subtree sizes. At worst, we will increase each of the $\Delta(i)$ for $i$ in the path by 1 since we may add the node $x$ to the larger subtree in every case. Again, as we showed in part (b), there are $O(\lg n)$ such nodes. The potential $\Phi(T)$ can therefore increase by at most $c\sum_{i \in path} 1 = O(c \lg n) = O(\lg n)$. So, the amortized cost for insertion and deletion is $O(\lg n) + O(\lg n) = O(\lg n)$.

3. **[10 points]** Off-Line Minimum

   Throughout this problem, your explanations should be as complete as possible, but they do not need to be formal. In particular, when asked to prove the correctness of the algorithm given in the problem, you do not need to use loop invariants; you may simply argue why a certain property should hold.

   (a) **[1 point]** Do problem 21-1(a) on page 519 of CLRS.

   **Answer:** For the sequence $4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5$ the *extracted* array will hold $4, 3, 2, 6, 8, 1$.

   (b) **[5 points]** Do problem 21-1(b) on page 519 of CLRS.

   **Answer:** We will show that the *extracted* array is correct by contradiction. Assume that the *extracted* array is not correct. Let $x = extracted[j]$ be the smallest value $extracted[j]$ for which the *extracted* array is incorrect. Let the correct solution reside in the array *correct*. Call $y$ the value of $correct[j]$. We have two cases, one where $x > y$ and one where $x < y$. We will prove that neither case can occur.

   Assume that $x > y$. Then the element $y$ can't appear in the *extracted* array, or it would have been the smallest value for which the *extracted* array was incorrect. Since we've already processed $y$ before processing $x$, it must have had a set value of $m + 1$. But, if $correct[j]$ was set to $y$, then $y$ must initially have been in some $K_i$ with $i \leq j$. Since $extracted[j]$ had no value yet when we processed $y$, then we couldn't have put $y$ in set $K_{m+1}$ since we only union with the set above us and we haven't unioned $K_j$ yet. Therefore, we can't have $x > y$.

   Assume that $x < y$. We argue that the element $x$ must appear in the *correct* array. Obviously $x$ must appear before the $j$th extraction in the original sequence since the OFF-LINE-MINIMUM algorithm never moves sets of elements backwards, it only unions with sets later than it. If we hadn't extracted $x$ by the $j$th extraction, then the optimal solutions should have chosen $x$ instead of $y$ for the $j$th extraction since $x$ is smaller. Therefore, the optimal solution must have extracted $x$ for some $i < j$. But, that means that $extracted[i]$ holds some $z > x$. By similar reasoning as above, we couldn't have moved $x$ past set $K_i$ since $extracted[i]$ would have been empty at the time $x$ was chosen. So, since we only union with sets above us and $K_i$ hasn't been unioned yet, we can't put $x$ in $extracted[j]$ before $extracted[i]$. Therefore, we can't have $x < y$.

   Since we have shown that we must have $extracted[j] = correct[j]$ for all $j$, the OFF-LINE-MINIMUM algorithm returns the correct array.

(c) **[4 points]** Do problem 21-1(c) on page 519 of CLRS.

**Answer:** We can efficiently implement OFF-LINE-MINIMUM using a disjoint-set forest. We begin by creating $n$ sets, one for each number, using the MAKE-SET operation. Then, we call UNION $n - m$ times to create the $K_j$ sets, the sequences of insertions between extractions. For each set $K_j$, we will also maintain 3 additional pieces of information, *number*, *prev*, and *next* in the representative element of the set (actually this information will be at all the elements, but it will only be maintained for the representative). *number* will correspond to $j$ and can easily be set with the initial creating of the sets. *prev* will point to the representative element of $K_{j-1}$ and *next* will point to the representative element of $K_{j+1}$. We can maintain these three properties through unions as follows: when we union two sets $j$ and $l$ where $l$ is the next set that still exists after $j$, we set *number* of the new representative equal to the maximum of the two representative *number*s, in this case, $l$. We set *prev* of the new set equal to *prev* of set $j$ and we set *next* of *prev* of $j$ equal to the new representative element. Similarly, we set *next* of the new set equal to *next* of set $l$ and we set *prev* of *next* of $l$ equal to the new representative element.

In the OFF-LINE-MINIMUM algorithm, each iteration of the for loop ($n$ iterations) will first call FIND-SET on $i$ (line 2). We use the *number* field of the returned representative as $j$. Then, in line 5, to determine the smallest $l$ greater than $j$ for which set $K_l$ exists, we can simply follow the *next* pointer from $j$, which takes constant time. In line 6, we call UNION again, at most $m$ times throughout the $n$ iterations. For each UNION call, we follow the procedure above for updating the *number*, *prev* and *next*, all of which take constant time. Therefore, we have a total of $n$ MAKE-SET operations, $n$ FIND-SET operations, and $n$ UNION operations. Since we have $3n$ total operations and $n$ MAKE-SET operations, we know by Theorem 21.13 that the worst-case running time is $O(3n\alpha(n)) = O(n\alpha(n))$.

4. **[23 points]** Articulation Points and Bridges

Throughout this problem, your explanations should be as complete as possible. You do not need to formally prove the correctness of any algorithm you are asked to give; however, when asked to prove some property of articulation points or bridges, your argument should be as formal as possible.

Notice that only parts (a)-(f) of problem 22-2 are assigned. Thus, do not worry about biconnected components for this problem.

If you are looking for the definition of a simple cycle, check appendix $B.4$.

(a) **[3 points]** Do problem 22-2(a) on page 558 of CLRS.

**Answer:** First we will prove the forward direction: If the root of $G_\pi$ is an articulation point, then it has at least 2 children in $G_\pi$. We will prove the contrapositive, the if the root has less than 2 children in $G_\pi$, then it is not an articulation point. If $r$ is the root and has no children, then it has no edges adjacent to it. Thus removing $r$ from $G$ can't disconnect the graph and $r$ is not

an articulation point. If $r$ has one child, then all other nodes in $G_\pi$ are reachable through the child. Therefore, removing $r$ from $G$ will not disconnect $G$ and $r$ is not an articulation point.

For the reverse direction, we need to show that if the root has at least 2 children in $G_\pi$ then the root is an articulation point. We will prove this by contradiction. Suppose the root has the two children $C_1$ and $C_2$ and that $C_1$ was explored first in the DFS. If the root is not an articulation point, then there exists a path between a node in $C_1$ and one in $C_2$ that does not include the root $r$. But, while exploring $C_1$, we should have explored this path and thus the nodes of $C_2$ should be children of some node of $C_1$. But, since they are in a separate subtree of the root, we know that no path can exist between them and thus $r$ is an articulation point.

(b) **[4 points]** Do problem 22-2(b) on page 558 of CLRS.

**Answer:** First we will prove the forward direction. Assume that $v$ is an articulation point of $G$. Let $C_r$ be the connected component of $G - v$ containing the root of the tree $G_\pi$. Let $s$ be a neighbor of $v$ that is not in $C_r$ (this neighbor must exist since removing $v$ must create at least two connected components) and $C_s$ be the connected component containing $s$. In $G_\pi$, all the proper ancestors of $v$ are in $C_r$, and all the descendants of $s$ are in $C_s$. Thus, there can be no edges between descendants of $s$ and proper ancestors of $v$.

To prove the backward direction, if such a vertex $s$ exists and there is no back edge from $s$ or any of it's descendants to a proper ancestor of $v$, then, we know that the only path from the root of the tree to $s$ goes through $v$. Therefore, if we remove $v$ from the graph, we have no path from the root of the tree $G_\pi$ to $s$, and we have disconnected the graph. Thus, $v$ is an articulation point.

(c) **[4 points]** Do problem 22-2(c) on page 559 of CLRS.

**Answer:** We can compute $low[v]$ for all vertices $v$ by starting at the leaves of the tree $G_\pi$. We compute $low[v]$ as follows:

$$low[v] = min(d[v], \min_{y \in children(v)} low[y], \min_{backedge(v,w)} d[w])$$

For leaves $v$, there are no descendants $u$ of $v$, so this returns either $d[v]$ or $d[w]$ is there is a back edge $(v, w)$. For vertices $v$ in the tree, if $low[v] = d[w]$, then either there is a back edge $(v, w)$, or there is a back edge $(u, w)$ for some descendant $u$. The last term in the min expression handles the case where $(v, w)$ is a back edge. If $u$ is a descendant of $v$ in $G_\pi$, we know that $d[u] > d[v]$ since $u$ is visited after $v$ in the depth first search. Therefore, if $d[w] < d[v]$, we also have $d[w] < d[u]$, so we will have set $low[u] = d[w]$. The middle term in the min expression therefore handles the case where $(u, w)$ is a back edge for some descendant $u$. Since we start at the leaves of the tree and work our way up, we will have computed everything we need when computing $low[v]$.

For each node $v$, we look at $d[v]$ and something related to all the edges leading from $v$, either tree edges leading to the children or back edges. So, the total running time is linear in the number of edges in $G$, $O(E)$.

(d) **[4 points]** Do problem 22-2(d) on page 559 of CLRS.

**Answer:** To compute the articulation points of $G$, we first run the depth first search and the algorithm from part (c). Depth first search runs in time $O(V+E)$ and since the graph is connected, we know $E > |V| - 1$ so this is simply $O(E)$. We also showed that the algorithm from (c) runs in time $O(E)$. Thus, we have calculated $low[v]$ for all $v \in V$. By part (a), we can test whether the root is an articulation point in O(1) time. By part (b), any non-root vertex $v$ is an articulation point if and only if it has a child $s$ in $G_\pi$ with no back edge to a proper ancestor of $v$. If $low[s] > d[v]$, then there must be a back edge to a proper ancestor of $v$. Otherwise, if there is an edge between a node $u$ that is not a proper ancestor of $v$ and $s$, and $u$ was visited before $v$, then we should have explored $s$ before visiting $v$.

So, if $v$ has a child $s$ in $G_\pi$ such that $low[s] \leq d[v]$, then $s$ has no back edge to a proper ancestor of $v$ and thus $v$ is an articulation point. We can check this in time proportional to the number of children of $v$ in $G_\pi$, so over all non-root vertices, this takes $O(V)$ time. Thus, the total time to find all the articulation points is $O(E)$.

(e) **[4 points]** Do problem 22-2(e) on page 559 of CLRS.

**Answer:** We will first prove the forward direction: if an edge $(u, v)$ is a bridge then it can not lie on a simple cycle. We will prove this by proving the contrapositive, if $(u, v)$ is on a simple cycle, then it is not a bridge. We know that if $(u, v)$ is a simple cycle, then there is a cycle $u \to v \to x_1 \to x_2 \cdots \to x_n \to u$, such that all of $u, v, x_i$ are distinct. If we remove the edge $(u, v)$, then any path which used to exist in the graph $G$ also exists in $G'$. We can prove this because any path which didn't include the edge $(u, v)$ obviously still exists. Any path which did include the edge $(u, v)$ can be modified to eliminate the edge $u \to v$ and include the path $u \to x_n \cdots \to x_1 \to v$ and similarly for the edge $v \to u$. Thus, $(u, v)$ is not a bridge. So, if $(u, v)$ is a bridge, then it is not on a simple cycle.

We will now prove the reverse direction and show that if an edge $(u, v)$ is not on a simple cycle, then it is a bridge. We will prove this by contradiction. Assume $(u, v)$ is not on a simply cycle but it is not a bridge. Let's say that we remove the edge $(u, v)$. Since $(u, v)$ is not a bridge, there is still a path connecting $u$ and $v$, $u \to x_1 \to x_2 \cdots \to x_n \to v$. Then, the edges $v \to u \to x_1 \cdots \to x_n \to v$ form a simple cycle. But, we assumed that $(u, v)$ wasn't on a simple cycle. So, $(u, v)$ must be a bridge.

(f) **[4 points]** Do problem 22-2(f) on page 559 of CLRS.

**Answer:** Any bridge in the graph $G$ must exist in the graph $G_\pi$. Otherwise, assume that $(u, v)$ is a bridge and that we explore $u$ first. Since removing $(u, v)$ disconnects $G$, the only way to explore $v$ is through the edge $(u, v)$. So, we only need to consider the edges in $G_\pi$ as bridges. If there are no simple cycles in the graph that contain the edge $(u, v)$ and we explore $u$ first, then we know that there are no back edges between $v$ and anything else. Also, we know that anything

in the subtree of $v$ can only have back edges to other nodes in the subtree of $v$. Therefore, we will have $low[v] = d[v]$ since $v$ is the first node visited in the subtree rooted at $v$. Thus, we can look over all the edges of $G_\pi$ and see whether $low[v] = d[v]$. If so, then we will output that $(parent[v]_{G_\pi}, v)$ is a bridge, i.e. that $v$ and its parent in $G_\pi$ form a bridge. Computing $low[v]$ for all vertices $v$ takes time $O(E)$ as we showed in part (c). Looping over all the edges takes time $O(V)$ since there are $|V| - 1$ edges in $G_\pi$. Thus the total time to calculate the bridges in $G$ is $O(E)$.

5. **[18 points]** Assorted Graph Problems

   Throughout this problem, your explanations should be as complete as possible, but they do not need to be formal. In particular, when asked to prove the correctness of the algorithm given in the problem, you do not need to use loop invariants; you may simply argue why a certain property should hold.

   (a) **[5 points]** Do problem 22.1-5 on page 530 of CLRS.

   **Answer:** The edge $(u, w)$ exists in the square of a graph $G$ if there exists a vertex $v$ such that the edges $(u, v)$ and $(v, w)$ exist in $G$. To calculate this efficiently from an adjacency matrix, we notice that this condition is exactly what we get when we square the matrix. The cell $M^2[u, w] = \sum_v M[u, v] \cdot M[v, w]$ when we multiply two matrices. So, if we represent edges present in $G$ with ones and all other entries as zeroes, we will get the square of the matrix with zeroes when edges aren't in the graph $G^2$ and positive integers representing the number of paths of length exactly two for edges that are in $G^2$. Using Strassen's algorithm or other more sophisticated matrix multiplication algorithms, we can compute this in $O(V^{2.376})$.

   Using adjacency lists, we need to loop over all edges in the graph $G$. For each edge $(u, v)$, we will look at the adjacency list of $v$ for all edges $(v, w)$ and add the edge $(u, w)$ to the adjacency lists for $G^2$. The maximum number of edges in the adjacency list for $v$ is $V$, so the total running time is $O(VE)$. This assumes that we can add and resolve conflicts when inserting into the adjacency lists for $G^2$ in constant time. We can do this by having hash tables for each vertex instead of linked lists.

   (b) **[4 points]** Use your result from part (a) to give an algorithm for computing $G^k$ for some integer $k \geq 1$. Try to make your algorithm as efficient as possible. For which $k$ is it asymptotically better to convert $G$ from one representation to another prior to computing $G^k$?

   **Answer:** To calculate the $G^k$ graph using the adjacency matrix representation, we can use the trick of repeated squaring. Thus, by first calculating $G^{k/2}$ for even $k$, and $G^{(k-1)/2}$ for odd $k$, we can solve the problem in time $O(V^{2.376} \lg k)$. For adjacency lists, we can do the same thing. If we calculate the $G^2$ graph, we can calculate the $G^4$ graph by running our algorithm from part (a) on the $G^2$ graph. Thus, to calculate $G^k$ using adjacency lists takes time $O(VE \lg k)$. Converting between the two representations takes time $O(V^2)$ which is asymptotically less than calculating $G^2$ in either case. Converting between the two represenations

therefore depends on how many edges you have relative to the number of vertices. If $E = o(V^{1.376})$ then you should convert to the adjacency list representation and otherwise you should convert to the matrix. Notice that the number of edges will keep changing for each $G^i$ so you may need to convert back and forth when calculating $G^k$.

(c) **[4 points]** Do problem 22.3-11 on page 549 of CLRS. How does your algorithm compare (in the asymptotic running time sense) with the algorithm given in class and in section 21.1 of CLRS for determining the connected components of $G$ using disjoint-set data structures?

**Answer:** We will first modify DFS to label the connected components.

```
DFS(G)
1   for each vertex u ∈ V[G]
2     do color[u] ← WHITE
3        π[u] ← NIL
4   time ← 0
5   k ← 0
6   for each vertex u ∈ V[G]
7     do if color[u] = WHITE
8        k ← k + 1
9        DFS-VISIT(u, k)
```

Lines 5, 8, and 9 are the ones which were added or changed. In DFS-VISIT, we will always call DFS-VISIT with the same value of $k$. In addition, after setting the color of $u$ to BLACK, we will set the connected component, $cc[u]$ to $k$.

Since the graph $G$ is undirected, two nodes $u$ and $v$ will only get the same connected component label if there is a path between them in the graph $G$.

The running time of this algorithm is the same as for DFS which is $O(V + E)$. The algorithm given in section 21.1 of CLRS runs in time $O((V+E)\alpha(V))$, which is asymptotically slower. However, $\alpha(V)$ is very small ($\leq 4$) for any reasonable size $V$, so the running times are comparable.

(d) **[5 points]** Do problem 22.4-2 on page 552 of CLRS.

**Answer:** We first run topological sort on the graph $G$. This takes time $O(V+E)$. We know that any path that runs between $s$ and $t$ must use only the vertices located between $s$ and $t$ in the topological sort. If there was a vertex $a < s$ in the topological sort, then there can't be a path from $s \rightarrow a$ in $G$. Likewise there can be no vertex $b > t$ on a path from $s$ to $t$. So, we can ignore all vertices $< s$ or $> t$ in the topological sort. Then, we can use dynamic programming to calculate the number of paths from $s$ to $t$. We will label each node from $s$ to $t$ with the number of paths from $s$ to that node. We start by labelling the node $s$ with a 1 since there is one path from $s$ to $s$ and by labelling all other nodes with 0. Then, for each node $i$ starting from $s$ in the sort, we calculate the number of paths from $s$ as

$$paths[i] = \sum_{(j,i)\in E} paths[j]$$

We finish when we calculate $paths[t]$ which we output as the answer. This algorithm is correct since all paths from $s$ to $i$ must only contain vertices between $s$ and $i$ in the topological sort. These we have calculated by the time we calculate $paths[i]$. Also, the predecessor to $i$ on any path from $s$ to $i$ must be such that there is an edge from $(pred, i)$. We sum over all possible predecessors to calculate $paths[i]$. For at most each vertex, we sum over the number of incoming edges. So, in total, we look at each edge once. Therefore, the running time of this step is $O(V + E)$. Thus, the total time for this algorithm is $O(V + E)$ which is linear in the size of the input.

6. **[18 points]** Minimal Spanning Trees

Throughout this problem, if you are arguing about the correctness of a minimal spanning tree algorithm, please be as formal as possible in proving the property the algorithm relies on for its correctness; however, you do not need to resort to loop invariants or similar formal methods to prove the correctness of the actual algorithm.

(a) **[6 points]** Do problem 23.1-11 on page 567 of CLRS.

**Answer:** If we decrease the weight of an edge $(u, v)$ not in $T$, then the new minimal spanning tree may now contain that edge. The algorithm to compute the new minimum spanning tree is to find the heaviest weight edge on the path from $u \to v$. If this edge weight is higher than the weight of edge $(u, v)$, then we delete this edge and add $(u, v)$. Else, we do nothing. The running time of this algorithm is the time taken to find the heaviest weight edge on the path $u \to v$. We can do this by running DFS from from $u$ till we hit $v$ since there is only one path from $u$ to $v$ in a tree. The running time is $O(V + E) = O(V)$ since this is a tree. The result is obviously a spanning tree. We will show that it is a minimum spanning tree.

We will do this by considering Kruskal's algorithm over the new graph. Kruskal's algorithm grows an MST by always adding the lowest weight edge that does not create a cycle. For all edges in the MST with weight less than $w'(u, v)$, we will take the exact same edges as we did before. When we consider the edge $(u, v)$, we have two choices, either we take the edge or we don't. If we don't take the edge, then $(u, v)$ must have created a cycle in the MST. All edges in the cycle must have weight less than $w'(u, v)$ or they wouldn't be in the MST already. The algorithm will then proceed as before. This corresponds to the above case where we don't modify the MST. If we take the edge $(u, v)$ then all further edges will be added in the same manner until we reach the one which creates a cycle with the edge $(u, v)$. This will be the highest weight edge in the original path from $u \to v$, which we won't add. All other edges will be added as before. This corresponds to the case where our algorithm deletes the highest weight edge on the path from

$u \to v$ and adds the edge $(u, v)$. Our algorithm produces the same spanning tree that Kruskal's algorithm produces, and it is therefore a minimum spanning tree.

(b) **[6 points]** Do problem 23.2-7 on page 574 of CLRS.

**Answer:** Let's assume that we add the new vertex and incident edges and initialize all the edge weights to $\infty$. Then, we can take any of the edges and add it to the original spanning tree to give a minimum spanning tree. Using the answer from part (a), we know that we can reduce the weight of this edge and the other edges one at a time, add the edge to the MST, and remove the edge with the highest weight in the newly created cycle. This will run in time $O(V^2)$ since there may be at most $V$ edges from the new vertex. However, we can do better by noticing that the only possible edges in the new MST are the ones in the old MST or the new edges we just added. There are a total of $|V| - 1$ edges in the old MST and a total of at most $|V|$ edges added. So, if we simply run either Kruskal's or Prim's algorithm on the graph with all the vertices but only these $|E| = 2|V| - 1 = O(V)$ possible edges, we will create the new MST in time $O(V \lg V)$, which is better than the $O(V^2)$ time.

(c) **[6 points]** Do problem 23.2-8 on page 574 of CLRS.

**Answer:** This algorithm does not compute the minimum spanning tree correctly. Suppose we have a graph with three nodes, $A, B, C$. Suppose also that the graph has three edges with the following weights: $w(A, B) = 1, w(B, C) = 2, w(C, A) = 3$. Let's say we partition the graph into the two sets $V_1 = \{A, C\}$ and $V_2 = \{B\}$. This partition satisfies the condition that $|V_1|$ and $|V_2|$ differ by at most 1. The edges sets will be $E_1 = \{(A, C)i\}$ and $E_2 = \emptyset$. So, when we recursively solve the subproblems, we will add the edge $(A, C)$. Then, when we select the minimum edge that crosses the partition, we will select the edge $(A, B)$ with weight 1. The total weight of our MST is $1 + 3 = 4$. However, the actual minimum spanning tree has edges $(A, B)$ and $(B, C)$ and weight $1 + 2 = 3$. Therefore, this algorithm fails to produce the correct minimum spanning tree.