2

```
0 def main(V, E, c, d):
1     children = find-children(V, E)
2     root = V[0]
3     result = find-flow(c, d, children, null, root)
4     if result = null
5         return "no such flow"
6     else
7         return result

0 def find-flow(c, d, children, parent, root):
1     flows = null
2     own-flow = d[root]

3     for each child ∈ {children[root]}:
4         child_flows = find-flow(c, d, children, root, child)
5         if child_flows = null
6             return null
7         incoming = child_flows[(root, child)]
8         if |incoming| > c[(root, child)]
9             return null
10        own_flow = own_flow + incoming
11        flows = merge(flows, child_flows)
12    if parent ≠ null
13        flows[(parent, root)] = own_flow
14    return flows
```

Variables & Input & Functions

main

V: a set of vertices in G, labeled as letters (space permitting). labels can be arbitrary, letters are just easy.

E: a set of edges in G, each edge $e=(u,v)$ $v,u \in V$ is a tuple of 2 vertices

C: capacity dictionary $E \to \mathbb{R}^+$, for some $e \in E$, $c[e] = x$, where x is e's cap.

d: demand dictionary $V \to \mathbb{R}$, for some $v \in V$, $d[v] = y$, where y is v's demand

find-children: this method takes a set of vertices V and a set of edges E from a tree G. It returns a dictionary $V \to \{V\}$ which maps a given vertex $v \in V$ to a list of its child vertices in tree G.*

find-flow

children: children dictionary $V \to \{V\}$, for some $v \in V$ children$[v] = \{c_1, c_2, c_3\}$ where $c_{1,2,3}$ are v's children in G

parent: this is a vertex which is the parent of "root", null if root is root of G

root: root vertex for all recursive and external invocations of find-flow

merge: this method merges 2 dictionaries, should not run into duplicate keys on merge. This isn't implemented, but can easily be implemented to run in $O(\min(s_1, s_2))$ where $\{s_1, s_2\}$ are the dictionary's sizes

Correctness

The algorithm traverses G in its entirety. By executing a depth first search, it recursively traverses a path until it hits a leaf node, then returns flow values back up the call stack. Because of its recursive nature, no flow for a given node (or its edges) is calculated until such values are calculated for its children, ensuring that each flow value is correct. Line (8) of find-flow checks down-stream (incoming) flow against the appropriate edge's respective capacity to satisfy the capacity constr-

Correctness (cont.'d)

aint.own_flow is initialized to root's demand. The algorithm then computes (recursively) all childrens' flows, updating own_flow to compute its own, assigning that flow to its parent if ~~the root~~ the parent is non-null. In the case that root is a leaf node, own_flow is initialized to root's demand, skips the for loop (as children[root] is null for leaf nodes), and updates the flows object w/ its own_flow value before returning the flows object. This algorithm is also correct for the trivial cases $|V|=1, |E|=0$ and $|V|=2, |E|=1$.

Complexity

(find-flow)

The algorithm runs in $O(n)$ linear time where $n=|V|$. Because the algorithm executes a depth-first search (DFS), it visits each node precisely once, performing constant-time arithmetic and conditional logic per visit, keeping the complexity linear. *Find-children is not implemented here, as Prof. Spielman stipulated "If you like, you may assume that the tree is given to you in this [root-child-leaf] form." Anyways, the method can be implemented in $O(n \log n)$ with an initial sort and subsequent linear scan.