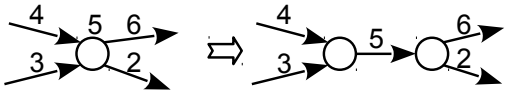
# Solutions to Self-study 1 Exercises

## CLRS4 24-1

***a.*** Given a flow network $G = (V, E)$ with capacities on vertices (as well as edges), it can be transformed into a "normal" flow network $G' = (V', E')$, where vertices do not have capacities. A vertex $v \in V$ is transformed to a pair of vertices $v'_1$ and $v'_2$ in $V'$ connected with an edge $(v'_1, v'_2) \in E'$, the capacity of which is set to the capacity of $v$. All ingoing edges of $v$ in $G$ become the ingoing edges of $v'_1$ in $G'$ and all outgoing edges of $v$ in $G$ become the outgoing edges of $v'_2$ in $G'$. Here is the picture illustrating the transformation.




The resulting graph has the same size asymptotically as the original, since $|V'| = 2|V|$ and $|E'| = |E| + |V|$.

***b.*** We transform the problem into a flow network with vertex capacities:

1. We introduce a source vertex $s$ and connect it to all the starting vertices.
2. We introduce a sink vertex $t$ and connect all the boundary vertices to $t$.
3. Each undirected edge of the grid is transformed into two *antiparallel* edges (see Section 24.1 of CLRS4). Then, each pair of antiparallel edges is modeled by introducing a new, auxiliary vertex as shown in Figure 24.2 in CLRS4. (Actually, this auxiliary vertex is not necessary, as we later transform the graph as described in part *a* and the two edges become no longer antiparallel.)
4. All edges and vertices (except for $s$ and $t$) are assigned the capacity of one.

A given edge/vertex of the grid belonging to some path is modeled by a positive flow through that edge/vertex in the flow network. Vertex capacities of one ensure that paths are vertex-disjoint. Once the flow network is constructed as above, it is transformed as described in part *a* and a max-flow algorithm such as Ford-Fulkerson is run. If the returned flow is equal to $m$ then the grid has an escape, otherwise (the flow is less than $m$) there is no escape.

The running time of the the flow-network construction and transformation steps is proportional to the size of the grid, $\Theta(n^2)$. The running time of Ford-Fulkerson is $\Theta(E|f|) = \Theta(Em) = \Theta(n^2 m)$. Thus the total running time is $\Theta(n^2 m)$.

## Exercise 2

**1.** When inviting $P$, only its "grandchildren" can be considered for invitation. In this case we invite all "grandchildren" and the resulting optimal friendliness of a participant list is $P.f + S_1.f + S_2.f + S_3.f = 15$.

If we choose not to invite $P$, then we have a choice of inviting or not inviting all of its immediate subordinates, $V$s. Then the resulting optimal optimal friendliness of a participant list is $V_1.f + S_2.f + S_3.f + V_3.f = 16$.

**2.** *What are the subproblems?* For this problem, the definition of a subproblem is very natural. Given a node $p$ in the organization structure, what is the optimal friendliness of the participant list covering the subtree rooted at $p$. *What are the choices that have to be considered when solving a subproblem?* The choice is

binary, either we invite $p$ or not. If we do not invite $p$, then the solution is the union of the solutions to all subproblems of immediate subordinates of $p$, otherwise, if we invite $p$, then the solution includes $p$ as well as all the solutions to the subproblems of "grandchildren". From the two choices, we choose the one that maximizes the sum of the friendliness ratings of the invitees.

This can be expressed as the following recurrence for the maximal friendliness rating of a participant list of a subtree rooted at $p$. Here $C_p$ is the set of all immediate subordinates of $p$, i.e., its "children". $C_p$ is empty if $p$ has no "children" and the sum over an empty set is zero.

$$F(p) = \max\left(\sum_{c \in C_p} F(c),\ p.f + \sum_{c \in C_p} \sum_{g \in C_c} F(g)\right).$$

Transforming this into a memoized dynamic programming algorithm is straightforward. The algorithm PARTICIPANT-LIST gets the root of the organization tree as an input and records in each node of the tree two additional fields: *fl*, the friendliness rating of the optimal participant list for this subtree, and *invite*, a flag indicating whether the person should be included in the participant list. Initially, *fl* should be set to $\infty$ for all nodes.

```
PARTICIPANT-LIST(p)
 1  if p.fl == ∞            // the solution for p is not yet computed
 2      sumc = 0
 3      sumg = 0
 4      c = p.left_child
 5      while c ≠ NIL
 6          sumc = sumc + PARTICIPANT-LIST(c)
 7          g = c.left_child
 8          while g ≠ NIL
 9              sumg = sumg + PARTICIPANT-LIST(g)
10              g = g.right_sibling
11          c = c.right_sibling
12      if p.f + sumg > sumc
13          p.fl = p.f + sumg
14          p.invite = TRUE
15      else
16          p.fl = sumc
17          p.invite = FALSE
18  return p.fl
```

Note that to print the optimal participant list, one has to remember to skip the "children" of an invited employee, even if some of them have the invite flags set to TRUE (these flags indicate whether an employee should be invited if *only the subtree rooted at that employee is considered*!).

Here is an alternative way of writing this algorithm. We introduce one more subproblem parameter, a boolean parameter $c$ specifying whether we invite the root of the subproblem or not. The recurrence then looks as follows:

$$F(p, c) = \begin{cases} p.f + \sum_{c \in C_p} F(c, \text{FALSE}) & \textbf{if } c = \text{TRUE}, \\ \sum_{c \in C_p} \max(F(c, \text{TRUE}), F(c, \text{FALSE})) & \textbf{else}. \end{cases}$$

For each of the nodes in the tree, the algorithm PARTICIPANT-LIST2 computes two fields: $fl\_with$ and $fl\_without$, friendliness rating of the participant list for the subtree when the employee is invited and when it is not invited. Although it is a recursive algorithm (as this is the most natural way to traverse a tree structure), the values are computed (lines 6 and 7) in a bottom-up order when the recursive calls unwind.

PARTICIPANT-LIST2($p$)
1 $p.fl\_with = f.p$
2 $p.fl\_wihout = 0$
3 $c = p.left\_child$
4 **while** $c \neq$ NIL
5 PARTICIPANT-LIST2($c$)
6 $p.fl\_with = p.fl\_with + c.fl\_wihout$
7 $p.fl\_without = p.fl\_without + \max(c.fl\_with, c.fl\_without)$
8 $c = c.right\_sibling$

When printing the optimal participant list, an employee is invited if $p.fl\_with \geq p.fl\_without$, and again one has to remember to skip the "children" of an invited employee.

The second algorithm is more elegant as it avoids a nested loop, but it is probably not as intuitive as the first one.

**3.** Analysing PARTICIPANT-LIST2 is easy—the tree is recursively traversed once. PARTICIPANT-LIST visits each node of the tree twice: from the parent and from the grandparent. On the second visit, it returns in line 1. Thus, the asymptotic running time of both variants of the algorithm is $\Theta(n)$.

**4.** Let $F^-(p)$ denote the friendliness rating of a participant list if we do not invite the root node $p$. Analogous, $F^+(p)$ is the friendliness rating if we invite the root node $p$. We prove the greedy choice property by comparing the upper bound of $F^-(p)$ with the lower bound of $F^+(p)$. Let $D_p$ be the set of all nodes in the subtree rooted at $p$, except $p$ itself (i.e., all subordinates of $p$).

If we do not invite $p$, then we can not achieve a larger friendliness rating than when we invite all of its subordinates: $F^-(p) \leq \sum_{d \in D_p} d.f$. If we do invite $p$, then even if we do not count any of its subordinates: $F^+(p) \geq p.f$. From the exercise we know that $p.f > \sum_{d \in D_p} d.f$. Combining the three inequalities proves that $F^-(p) < F^+(p)$, i.e., inviting $p$ will always result in a friendlier participant list.

The greedy algorithm is then a recursive traversal down the tree starting with inviting the root and then skipping every second level.

**5.** The running time is $\Theta(n)$, the same asymptotically as for the dynamic programming, but the greedy algorithm will be faster as it touches every node only once and does not compute anything (does not add up friendliness ratings nor compare them).