1. We give a dynamic programming algorithm.

Step 0: *Describe the recursive structure of sub-problems.*

Consider an input $E = f(e_1, e_2, \ldots, e_n)$. Note that if any e_i is a variable, then it has *no* effect on the evaluation time. So w.l.o.g., let $e_i = g_i(\ldots)$ for each i.

Any optimal solution for this input has the following form, where $i_1, i_2, \ldots, i_n$ is a permutation of $[1, 2, \ldots, n]$:

- time 0: call $f()$
- time 1: f calls $g_{i_i}()$
- time 2: f calls $g_{i_2}()$, ...
- ...
- time n: f calls $g_{i_n}()$, ...
- ...

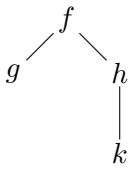
The overall evaluation time for this solution is $T(f) = \max\{1 + T(g_{i_1}), 2 + T(g_{i_2}), \ldots, n + T(g_{i_n})\}$, where $T(g_{i_j})$ is the evaluation time for input $g_{i_j}(\ldots)$ (performing the steps in the call to g_{i_j} in the same order as in the overall solution).

Note that in this solution, it is possible that some of the calls to functions g_i are performed in a sub-optimal manner — because they may not contribute to raise the overall maximum. But is it *not* possible for *every* call to g_i to be done in a sub-optimal manner — otherwise, we could improve on every one of those calls and lower the overall completion time. This means that an optimal solution to the original problem can always be obtained from optimal solutions to the sub-problems $g_1, g_2, \ldots, g_n$.

Also, the same argument as in Question 1 of Assignment 1 shows that ordering the calls to the g_i's so that $T(g_{i_1}) \geqslant T(g_{i_2}) \geqslant \ldots \geqslant T(g_{i_n})$ minimizes $T(f)$.

Step 1: *Define an array that stores optimal values for arbitrary sub-problems.*

There is no simple linear structure to specify sub-problems. However, every function call expression E can be represented as a tree, with one node for each function symbol in E, where the children of node f are exactly the functions called directly by f — variables are ignored. For example, the expression $f(g(x), h(y, k()))$ is represented by the tree on the right.



For each function symbol f in the expression E, define $T[f]$ to be the minimum evaluation time for the sub-expression rooted at f.

Step 2: *Give a recurrence relation for the array values.*

For every leaf f in the tree representation for E, $T[f] = 1$.

For every internal node $f(g_1, \ldots, g_n)$ in the tree, $T[f] = \max\{1+T[g_{i_1}], \ldots, n+T[g_{i_n}]\}$, where $i_1, \ldots, i_n$ is a permutation of $[1, \ldots, n]$ such that $T[g_{i_1}] \geqslant \ldots \geqslant T[g_{i_n}]$.

Step 3: *Write a bottom-up algorithm to compute the array values, following the recurrence.*

```
Construct the tree representation for E — call it R
Perform a post-order traversal of R — for each node f:
    if f is a leaf:
        T[f] ← 1
    else:
        let g_1, g_2, ..., g_k be the children of f, ordered so that T[g_1] ⩾ T[g_2] ⩾ ... ⩾ T[g_k]
        T[f] ← max{1 + T[g_1], 2 + T[g_2], ..., k + T[g_k]}
```

The running time of this algorithm is $\Theta(n \log n)$, where n is the number of nodes in the tree representation of E: each node is visited once by the post-order traversal, and each node's children are sorted (which takes no more than $\mathcal{O}(\log n)$ each time). To see that the bound is tight, consider the expression $E = f(g_1(), g_2(), \ldots, g_k())$: sorting the children of f takes time $\Omega(k \log k) = \Omega(n \log n)$ since $n = k + 1$.

Step 4: *Use the computed values to reconstruct an optimal solution.*

Starting at the root of R, for each node f in R with children $g_1, \ldots, g_k$, simply call the g_i's in non-increasing order of $T[g_i]$.

This takes only a linear amount of time, in addition to the time spent to compute the T values.

2. **Step 0:** *Describe the recursive structure of sub-problems.*

In every optimal solution $(\ell_{i_0+1}, \ldots, \ell_{i_1}); (\ell_{i_1+1}, \ldots, \ell_{i_2}); \ldots; (\ell_{i_{k-1}+1}, \ldots, \ell_{i_k})$, the division of the first $k-1$ groups (the ones that correspond to input $\ell_1, \ldots, \ell_{i_{k-1}}$) must be done optimally — otherwise, we could replace the first $k-1$ groups to get a better overall solution.

Step 1: *Define an array that stores optimal values for arbitrary sub-problems.*

For $j = 0, 1, \ldots, n$, let $A[j]$ denote the value of an optimal solution to the problem with input $\ell_1, \ldots, \ell_j$.

Step 2: *Give a recurrence relation for the array values.*

$A[0] = 0$ (there are no solutions to consider).

For $j = 1, 2, \ldots, n$, $A[j] = \min\left\{A[i] + (L - \ell_{i+1} - \cdots - \ell_j)^2 : 0 \leqslant i < j \text{ and } \ell_{i+1} + \cdots + \ell_j \leqslant L\right\}$ (consider every possible last group).

Step 3: *Write a bottom-up algorithm to compute the array values, following the recurrence.*

```
A[0] ← 0
for j ∈ [1, 2, ..., n]:
    A[j] ← A[j − 1] + (L − ℓ_j)^2
    for i ∈ [j − 2, j − 3, ..., 1] and while ℓ_{i+1} + ℓ_{i+2} + ··· + ℓ_j ⩽ L:
        if A[i] + (L − ℓ_j − ··· − ℓ_{i+1})^2 < A[j]:
            A[j] ← A[i] + (L − ℓ_j − ··· − ℓ_{i+1})^2
```

Runtime: $\Theta(n^2)$ for the two nested loops, in the worst-case.

Step 4: *Use the computed values to reconstruct an optimal solution.*

We need a second array $B[j]$ to store the index i such that $A[j] = A[i] + (L - \ell_j - \cdots - \ell_{i+1})^2$.

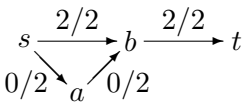
```
A[0] ← 0
B[0] ← 0
for j ∈ [1, 2, ..., n]:
    A[j] ← A[j − 1] + (L − ℓ_j)^2
    B[j] ← j − 1
    for i ∈ [j − 2, ..., 1] and while ℓ_{i+1} + ··· + ℓ_j ⩽ L:
        if A[i] + (L − ℓ_j − ··· − ℓ_{i+1})^2 < A[j]:
            A[j] ← A[i] + (L − ℓ_j − ··· − ℓ_{i+1})^2
            B[j] ← i
```

Now, we can reconstruct the solution one group at a time, starting from the end and working backward.

```
S ← ∅
j ← n
while j > 0:
    S ← S ∪ {(ℓ_{B[j]+1}, ..., ℓ_j)}
    j ← B[j]
return S
```

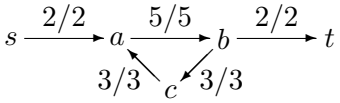
Runtime: $\Theta(n^2)$ to compute the values in $A[\,]$ and $B[\,]$ (reconstructing the solution takes only $\Theta(n)$).

3. (a) Consider the network N on the right, with the flow f indicated in the diagram, and let $e_0 = (s, b)$. Reducing the capacity of e_0 to 1 does not decrease the maximum flow because additional flow can be routed through (s, a) and (a, b).



(b) Let $e_0 = (a, b)$. Intuitively, we want to find a "reducing path" from s to a and another "reducing path" from b to t, to rebalance the flow in the network after it has been reduced by 1 unit through edge e_0. Then, we can search for an augmenting path in N', in case there is a way to re-route the flow.

There is one technical twist we must deal with. In general, all flow coming into a node a can be traced back to the source s, and similarly, all flow coming out of a node b can be followed forward to the target t. However, it is possible for a network to contain a directed cycle of edges where "phantom flow" is circulating: a certain amount of flow that simply goes around the cycle, without contributing anything to the overall flow value. For example, $a \to b \to c$ is such a cycle in the network below, with 3 units of "phantom flow:"



This is the reason for the first if-statement in the algorithm below.

Given N, f, and $e_0 = (a, b)$:
  Set $f'(a, b) = f(a, b) - 1$.
  Run BFS starting from a, using only backward edges (x, y) with $f(x, y) > 0$ or
    forward edges (y, x) with $f(y, x) < c(y, x)$ — (looking for a "reducing path").
  If b is reached: # *we have found a cycle that contains* (a, b)
    Adjust the flow on every edge from a to b:
      Set $f'(x, y) = f(x, y) - 1$ for every backward edge (x, y) with $f(x, y) > 0$.
      Set $f'(y, x) = f(y, x) + 1$ for every forward edge (y, x) with $f(y, x) < c(y, x)$.
  Else: # *there is no cycle that contains* (a, b)
    # *There must be a path from* a *to* s.
    Adjust the flow on every edge from a back to s:
      Set $f'(x, y) = f(x, y) - 1$ for every backward edge (x, y) with $f(x, y) > 0$.
      Set $f'(y, x) = f(y, x) + 1$ for every forward edge (y, x) with $f(y, x) < c(y, x)$.
    # *There must also be a "reducing path" from* b *forward to* t.
    Run BFS starting from b, using only forward edges (x, y) with $f(x, y) > 0$ or
      backward edges (y, x) with $f(y, x) < c(y, x)$ — (looking for a "reducing path").
    Adjust the flow on every edge from b to t:
      Set $f'(x, y) = f(x, y) - 1$ for every forward edge (x, y) with $f(x, y) > 0$.
      Set $f'(y, x) = f(y, x) + 1$ for every backward edge (y, x) with $f(y, x) < c(y, x)$.
    # *Now, check if the flow can be re-routed.*
    Run BFS on N' to find an augmenting path, and if one is found, do the augmentation.

The correctness of the algorithm follows from the reasoning that precedes the algorithm.

The algorithm runs in worst-case time $\Theta(n + m)$ (linear in the size of the network): BFS takes linear time in the size of the input graph and is executed at most three times, and doing each reduction and augmentation also takes linear time (traversing a single path).

(c) Consider the network $N = s \xrightarrow{2/2} a \xrightarrow{2/2} t$ with the flow f indicated in the diagram, and let $e_0 = (s, a)$. Increasing the capacity of e_0 to 3 does not increase the maximum flow because no additional flow can be pushed on edge (a, t).

(d) Let us state the facts. We have a maximum flow in network N. This means that N contains *no* augmenting path (by the Ford-Fulkerson Theorem). The only way for the maximum flow to increase in N' is to have an augmenting path in N'. And the only difference between N' and N is the increase in the capacity of edge e_0.

So we can find a new maximum flow by looking for an augmenting path that uses edge e_0. If there are no such paths, then we know that there is no augmenting path in N' and flow f is already maximum. If there is one augmenting path, we can augment the flow along this path. But then we are done: there

can be no other augmenting path because edge e_0 will be back at full capacity and everything else is the same as in N.

Given N, f, and e_0:
    Run BFS on N' to find an augmenting path.
    If we find one augmenting path P: # *P will contain e_0*
        Augment f along P.
    # *Else: do nothing — there is no other augmenting path in N.*

The correctness of the algorithm follows from the reasoning that precedes the algorithm.

The algorithm runs in worst-case time $\Theta(n + m)$ (linear in the size of the network): BFS takes linear time in the size of the input graph, and doing one augmentation also takes linear time.