Kate Lassiter

**For all algorithms that you are asked to "give" or "design", you must do all of the following to get full credit: (a) Describe your algorithm clearly in English. (b) Give pseudocode. (c) Argue correctness, even if you don't give a formal proof and give a convincing argument instead. (d) Give with an explanation the best upper bound that you can for the running time.*
**If you give a Dynamic Programming algorithm, the above requirements are modified as follows: (a) Clearly define the subproblems in English. (b) Explain the recurrence in English. (This counts as a proof of correctness; feel free to give an inductive proof of correctness too for practice but points will not be deducted if you don't). Then give the recurrence in symbols. (c) State boundary conditions. (d) Analyze time. (e) Analyze space. (f) If you're filling in a matrix, explain the order to fill in subproblems in English. (g) Give pseudocode.*

1. (35 points) On input three sequences X, Y, Z of sizes m, n, p respectively, give an efficient algorithm to compute the length of their longest common subsequence. You should also give an algorithm to output a longest common subsequence. For example, if X = abcde, Y = abde and Z = abce, then the longest common subsequence is abe and its length is 3. (You may think of m, p as being Θ(n).)

Part 1: Length of longest subsequence

(a) Clearly define the subproblems in English.

1. If the given letter of all three strings match, then the length of the longest subsequence is 1 + the length of the longest subsequence found up to that point. Any other common subsequence will be found in the rest of the 3 strings not considering that letter.
2. If they do not match, then the length of the longest subsequence is equal to the maximum subsequence calculated for the three strings if one letter was removed, i.e carrying forward the longest common subsequence found thus far. It is the maximum length subsequence if the current string had a letter removed and the other two strings were left as they were. In this way all possible combinations would have to be calculated. Using a dynamic programming table to store these combinations found at a prior step avoids having to recalculate these numbers.

(b) Explain the recurrence in English. Then give the recurrence in symbols.

Example:
String1 = "LRC"
String2="GBC"
String3="BCD"

For the three strings, if the letter found at the first index is not equal across strings, longest common subsequence can be found in three ways:
["_**RC**","GBC","BCD"]
["LRC","_**BC**","BCD"]
["LRC","GBC","_**CD**"]
If they were the same, then the length of the longest common subsequence is 1 and any other common subsequence would be found in the rest of the strings:
["_**RC**","_**BC**","_**CD**"]

All these subproblems can be solved to find the longest common subsequence. In order to speed up time complexity, dynamic programming can be used to store solutions to overlapping subproblems, as subsequence combinations are found multiple times and there are only m x n x p unique problems.

OPT(m,n,p) = longest common subsequence between $x_1...x_i$, $y_1...y_j$,$z_1....z_k$

For 1=i,j,k… n,m,p

1. If (i,j,k) ∈ L, 1 + OPT(i-1,j-1,k-1).
2. If (i,j,k) not ∈ L, MAX(OPT(i−1,j,k),OPT(i,j-1,k),OPT(i,j,k-1))}

(c) State boundary conditions.

OPT(0,n,p) = 0

OPT(m,0,p) = 0

OPT(m,n,0) = 0

(d) Analyze time & space.

O(mnp) time complexity as this involves three nested for loops of lengths n,m,p. O(mnp) space complexity to store the m x n x p dynamic programming table.

(g) Give pseudocode.

```
max_common_subsequence(s1,s2,s3):
    ################## Part 1 #####################
    #First finding the length of the subsequence
    m = s1.length, n = s2.length, p = s3.length
    Initialize zero array M of size m x n x p
    if m = 0 or n = 0 or p = 0 do
        return 0,0
    for i =1 to m do
        for j=1 to n do
        for k=1 to p do
                if i or j or k = 1 do
                    M[i][j][k] = 0
                else if s1[i] = s2[j] = s3[k] do
                    M[i][j][k] = 1 + M[i-1][j-1][k-1]
                else do
                    M[i][j][k] = max(M[i-1][j][k], M[i][j-1][k],
                                     M[i][j][k-1])
            end for
        end for
    end for

    max_length_subsequence=M[m][n][p]

    ################## Part 2 #####################
     #Then returning the actual subsequence
     Initialize array of zeros of size max_length_subsequence as sequence
     Initialize index = 0 to fill in sequence sequentially
      while m > 1 and n > 1 and p > 1 do
        current_max = M[m][n][p]
        if current_max = M[m-1][n][p] do
          decrement m by 1
        else if current_max = M[m][n-1][p] do
          decrement n by 1
        else if  current_max = M[m][n][p-1] do
          decrement p by 1
        else do
           sequence[max_length_subsequence-index]=s1[m-1]
```

```
                decrement m by 1
                decrement n by 1
                decrement p by 1
                 Increment index by 1
            return  max_length_subsequence, sequence
```

Part 2: Return actual longest subsequence string

(a) Description:

1. While we haven't reached the end of any of the strings, if the longest subsequence length is equal to the longest subsequence in the prior row/column/vector (n,m,p), move further back in that direction because this isn't the letter that is part of the sequence. When a point is found where the longest subsequence value changes in direction m, n and p, this is the character that is part of the longest subsequence.

(b) Analyze time.

O(mnp) as this involves iterating in the worst case through all strings 1, 2 and 3.

(g) Give pseudocode:

Shown above.

2.(40 points) A flow network with demands G = (V, E, c, d) is a directed capacitated graph with potentially multiple sources and sinks, which may have incoming and outgoing edges respectively. In particular, each node v ∈ V has an integer demand dv; if dv > 0, v is a sink, while if dv < 0, it is a source. Let S be the set of source nodes and T the set of sink nodes. A circulation with demands is a function f : E → R+ that satisfies

(a) capacity constraints: for each e ∈ E, 0 ≤ f(e) ≤ ce.

(b) demand constraints: For each v ∈ V , f in(v) − f out(v) = dv.

We are now concerned with a decision problem rather than a maximization one: is there a circulation f with demands that meets both capacity and demand conditions? i. (10 points)

1. **Derive a necessary condition for a feasible circulation with demands to exist.**
   Because only sources have dv<0 and sinks dv>0, and dv=f_{in}(V) - f_{out}(V), that means that f_{in}(V) - f_{out}(V) = 0 between the S source nodes and T sinks. So the sum of the demand out of the sink nodes, for all t in T, equals the negative sum of the demand out of the source nodes, s in S.
2. **Reduce the problem of finding a feasible circulation with demands to max flow.**

(a) Inputs to the two problems.
Reduction R: The graph G (input to algorithm),
Algorithm for Y: Constructed flow network G′ (input to max flow black box)

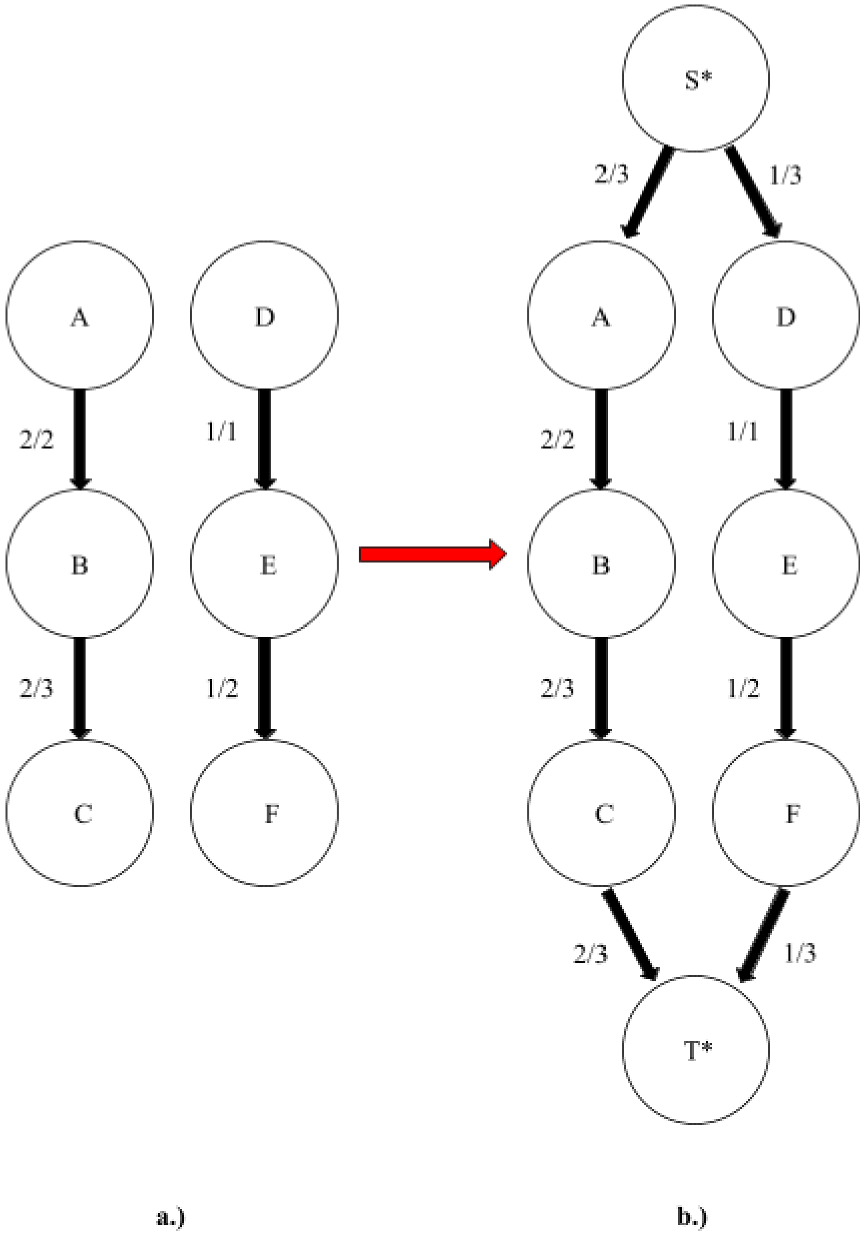
(b) Describe reduction transformation, argue polynomial time

1. Add a source s*
2. Add (s*, s) edges for all s ∈ S
3. c_e(s*,s) = -d(s) to preserve dv=f_{in}(V) - f_{out}(V)=0 for non-source/sink nodes. Need to make sure that the flow coming in from the new source s* is equal to the flow going out of the original source.
4. Add a sink t*
5. Add (t*, t) edges for all t ∈ T.

Kate Lassiter

6. $c_e(t,t^*) = d(t)$ in order to preserve $dv=f_{in}(V) - f_{out}(V)=0$. Need to make sure that the flow coming into the new sink t* is equal to the flow into the original sink.
   Given the flow network, solve the decision problem for max flow by transforming G as described and running max flow that outputs "yes" or "no". If there is a maximum flow found in the equivalent network equal to the target value, the sum of the demand out of the sink nodes, for all t in T, output "yes". Else output "no". If this flow exists in the modified graph, then there is a circulation meeting these constraints for the original graph. Thus the problem is correctly solved through reduction. This is clearly polynomial time because it simply involves adding at most n-1 edges to source and the sink and setting demand values, then running max flow which has been shown to run in $O(nm^2)$ when performed with BFS.

(c) Prove equivalence of the original and the reduced instances



Kate Lassiter

1. ***If there is a feasible circulation of max flow f in the original network G, there is a feasible circulation of max flow f' of the same value as f in G'.***

For a feasible flow to exist in G, the flow in and out of all vertices must equal zero except for the source and sink nodes. All capacities must be in the bounds as well. Because the flow passed into/out of each of the individual paths in G' is just the flow that initially came out/into them, the solution to G equals the solution to G'. The additional edges in G' help meet flow conservation constraints. The max flow between all the paths can be calculated as the flow into t* in G'. This is the amount that would be sent through each of the sources of the individual paths in G and arrive at the sinks given feasible circulation. This amount now flows from the source s* in G' to the rest of the paths and arrives at t*. The flow out of s* in G' equals the total demand of sinks in G in order for feasible circulation in G. This is the max flow because there's no further way to increase the flow through G' as the total capacity of s* in G' is met.

In figure a, there is a feasible circulation for path ABC of flow = 2 and DEF of flow = 1. This satisfies both the demand and capacity constraints, $dv=f_{in}(V) - f_{out}(V)=0$ for all nodes except the sources and sinks and the maximum flow that can be pushed that is $0<f(e)<c(e)$ for both paths are pushed. Given there is a feasible circulation in this original graph, there is also one for the augmented graph G' as in figure b. Here the flow for ABC remains 2 and DEF is still 1. $dv=f_{in}(V) - f_{out}(V)=0$ for all nodes in the original graph, and all are within $0<f(e)<c(e)$, so there is clearly still feasible circulation. The only modification is essentially an extra edge from the source to both of the separate paths ABC and DEF that is equal to the negative demand of the original source for those graphs. Only if there was feasible flow in the original network would there be feasible flow in G' because G' is simply the same graph G, only with additional source and sink nodes that are guaranteed to meet demand and capacity constraints.

2. If there is a feasible circulation of max flow f' in the transformed network G', there is a feasible circulation of max flow f of the same values as f' in G.

The max flow f' in G' when pushed across the edges of the original network G will result in a max flow in G. Capacity constraints are satisfied in G' and thus also in G because the edges are the same. The amount pushed from the source in G' to the original G source nodes A and D are the same as they were originally, and the amount pushed out to the sink in G* is also the same as G, thus demand constraints are satisfied. So the max flow of G* is equal to the max flow of G when the same flows are pushed to G as in G'.

3. 30 points) In many applications, on top of the node demands introduced in the previous problem, the flow must also make use of certain edges. To capture such constraints, consider the following variant of the previous problem. You are given a flow network with demands and lower bounds G = (V, E, c, d, `) where every edge e has an integer capacity ce, and an integer lower bound `e ≥ 0. A circulation f must now satisfy `e ≤ f (e) ≤ ce for every e ∈ E, as well as the demand constraints. Determine whether a feasible circulation exists.

a. **Derive a necessary condition for a feasible circulation with demands to exist.**

Same as problem 2, $f_{in}(V) - f_{out}(V) = 0$ between the S source nodes and T sinks. So the sum of the demand out of the sink nodes, for all t in T, equals the negative sum of the demand out of the source nodes, s in S. The addition here is lower bound constraints for the edges. We need to meet both demand and capacity constraints. Meeting capacity constraints is easy by just setting it equal to f(e) >= L. This however throws off demand constraints, where $dv=f_{in}(V) - f_{out}(V)$ may no longer equal zero for some vertices. To have feasible circulation, no flow can be less than the lower bound.

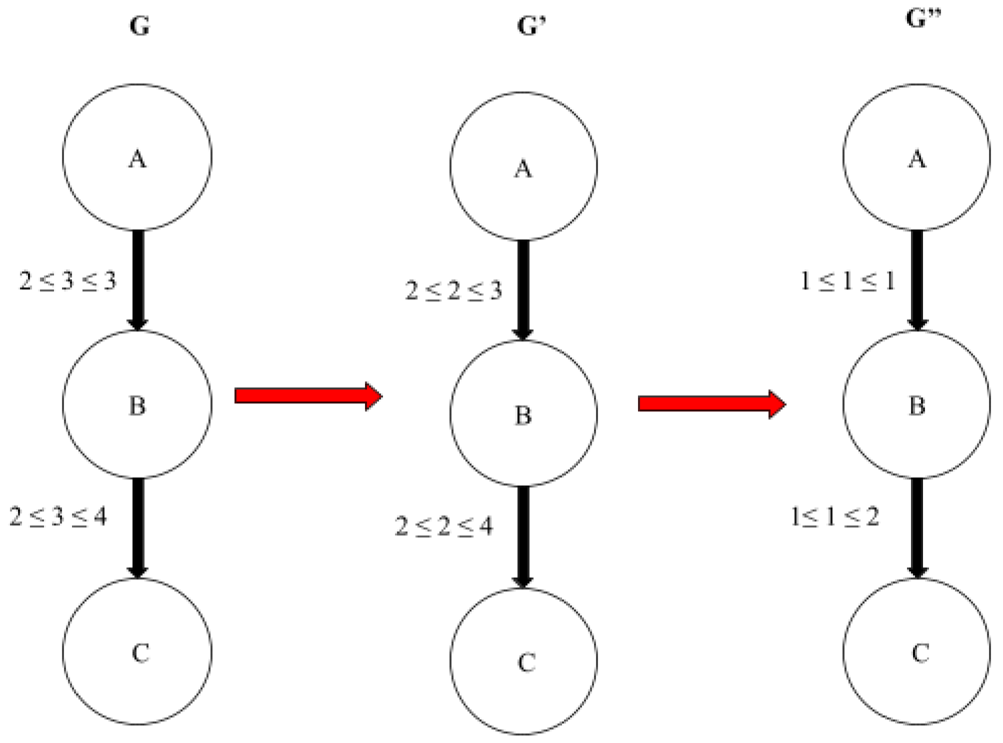
b. Inputs to the two problems.
Reduction R: The graph G (input to algorithm),
Algorithm for Y: Constructed flow network G′ (input to max flow black box)

c. Describe reduction transformation, argue polynomial time
   1. Make the same modifications to the graph G' as in question two.
   2. Demand constraints:
      a. First set f(e) =l(e) for every edge. Calculate the demand for every node as f_{in}(V) - f_{out}(V). This is the minimum possible flow through this vertex to satisfy the lower bounds.
      b. For every vertex, subtract this minimum possible flow in and out from the current demand. Now the flow in and out of every node can not be less than the lower bounded flow. The flow in/out of the vertex is now equal to the remaining flow that can be utilized during Ford Fulkerson.
   3. Capacity Constraints:
      a. Subtract the lower bound from the max capacity for each edge. This means that during the course of the algorithm the capacity of every edge will only be the flow leftover above this bound. This is because no more than c(e)-l(e) units can be pushed forward or backwards during FF without violating the lower bound. This satisfies capacity constraints.

This transformation happens in polynomial time, as it requires iterating over all the edges once to remove the lower bound, setting the flow equal to the lower bound over n nodes, and then calculating the difference between the n demands between the two graphs.

d. Prove equivalence of the original and the reduced instances

Kate Lassiter



a.)

D(v)
A = -3
B = 0
C = 3

i.)

D(v')
A = -2
B = 0
C = 2

ii.)

$D(v^{New}) = D(v) - D(v')$
A = -3 + -2 = -1
B = 0
C = 3 - 2 = 1

$c(e^{New}) = c(e) - l(e)$
AB = 3 - 2 = 1
BC = 4 - 2 = 2

1. If there is a feasible circulation of max flow f in the original network G, there is a feasible circulation of max flow f’ of the same value as f in G’.

For feasible circulation to exist in the original graph G, then the flow in and out of all vertices is zero except for sources and sinks, and the flow is greater than or equal to the lower bound for all edges. If the lower bound was simply zero, max flow could be calculated as described above using Ford Fulkerson. If there is feasible circulation in G, then G’ will contain only the flow above the lower bound in G that is at least as large as the minimum required flow. G’ cannot have a flow less than the lower bounds of G. Now Ford Fulkerson can be ran on this graph to produce an equivalent output, that when added to the lower bound for each edge, would have resulted if Ford Fulkerson was applied to the original graph G. If there was not a feasible flow in the original G, then we would not arrive at a feasible circulation in G’. Even if the flow was able to meet lower bounded capacity constraints of individual edges, the demand constraint would never be zero if the minimum sum of the flow out of edges of a vertex was greater or less than the minimum sum of the edges into the vertex for non source/sink nodes.

In the example G in figure a, the max flow here is 3, with edges AB and BC lower bounded by 2. The transformed graph G' first sends the lower bounded flow through the path as shown in figure i.). Then the difference between the current demand and minimum bounded demand (DV^{new}) is set as the flow through the path in G'', with max capacity (ce^{new}) equal to the leftover capacity that can be pushed above the lower bound in figure ii.). By adding the lower bound to the flow across the edges found in G'', this equals the max flow for G, which equals 3. The flow through the path in G shown in figure a satisfies both the capacity lower bounded constraints and also $dv=f_{in}(V) - f_{out}(V)=0$ for all nodes except the source and sink nodes. Thus the two are equivalent.

2. If there is a feasible circulation of max flow f' in the transformed network G', there is a feasible circulation of max flow f of the same values as f' in G.

Given there is feasible flow in G', this implies there is a flow that meets both demand constraints with $dv=f_{in}(V) - f_{out}(V)=0$ for all nodes except the source and sink nodes and capacity constraints above the minimum bound for every edge. Because G' already satisfies the lower bound constraints of G, the max flow found in G' will equal the max flow in G just with the lower bound considered. After the lower bounded flow is subtracted from the current flow in G, the rest of the flow can be utilized as a valid circulation in G'. By setting the capacity of each edge in G' equal to the current capacity in G minus the lower bound, no less capacity than the lower bound will be utilized while the max flow is calculated. Thus the demand and capacity constraints will both be satisfied in G' if and only if the constraints would also be satisfied in G. If we add the flow found in G' to the lower bounds for each edge, we will have the max flow in G.