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. ***(25 points) Given an undirected (unweighted) graph $G = (V, E)$ and two nodes $s, t \in V$, give an $O(n+m)$ algorithm that computes the number of shortest s-t paths in G.***

Algorithm Description:
By slightly tweaking BFS to keep track of the number of paths from s to $v_{i,...}v_{n,}$ initialized to 1 and then as the number of paths to the parent node, and then checking if t has been discovered by another node in the prior layer, you can find the total number of shortest s-t paths.

Pseudocode:

```
num_shortest(G=(V,E), s E V, target):
    Array discovered[V] initialized to 0
    Array dist[V] initialized to infinity
    Array parent[V] initialized to NIL
    Queue q
    discovered [s] = 1
    Dist[s] = 0
    target=1
    parent[s]=NIL
    num_paths[s]=1
    enqueue(q,s)
    while size(q)>0 do
        u=deque(q)
        for (u,v) E E do
            if discovered[v] == 0 then
                Discovered[v] = 1
                dist[v]=dist[u]+1
                Parent[v] = u
                num_paths[v] = num_paths[u]
                enqueue(q,v)
            else if dist[v] == dist[u] +1:
                num_paths[v]=num_paths[v]+num_paths[u]
        end for
end while
return(num_paths[target])
```

Correctness:
As we construct a BFS tree, only nodes that appear in the same layer can be the shortest path from the source to a node in that layer. This is because if there was a shorter way to reach this node from the source, it would have been added to BFS output at a higher up layer, because BFS

explores all its closest neighbors in every layer. Thus, if we encountered a path to t on a given layer, and there was a node on the prior layer that also pointed to t, these are equi-distant, shortest s-t paths.

Base case:

This base case is k=1 single node. If there is only one node, it is added to the queue to be explored, initialized with the number of shortest paths equal to 1. When arriving at the for loop, no neighbors are found, the size of the queue becomes zero, and the number of paths returned is 1.

Hypothesis:

All the number of shortest paths from the source have been calculated.

Loop variant:

Consider the case where k=4 nodes, with an edge from the source to node 3 and node 9. These nodes will be added to the queue, discovered, and have the number of shortest paths to these nodes set to 1. If from node 3, node t is discovered for the first time, it will be added to the queue and have its distance from the source node set to 2, with the number of paths to t set to 1. If t can be discovered from node 9 as well when it is popped from the queue, the number of paths will iterate by 1 because now 2 shortest paths can be found. Thus the loop variant is correct

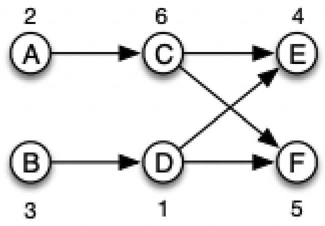
Termination:

The loop ends when all nodes have been explored, and the number of paths of node t is returned.

Running Time:

Because this is just a slight constant time modification of BFS, which has been proven to run in O(n+m), that's the runtime of this algorithm.

2. ***(20 points) You are given a directed graph G = (V, E) in which each node u ∈ V has an associated price pu which is a positive integer. Define the array cost as follows: for each u ∈ V , cost[u] = price of the cheapest node reachable from u (including u itself) For instance, in the graph below (with prices shown for each vertex), you can confirm by inspection that the cost values of the nodes A, B, C, D, E, F are 2, 1, 4, 1, 4, 5 respectively. Your goal is to design an algorithm that fills in the entire array cost. Give a linear-time algorithm that works when the input is a Directed Acyclic Graph (DAG).***



Algorithm Description:

This is a classic example of DFS, you want to explore all the nodes in the graph. While you're running DFS, every time a task finishes, add this to a list of nodes (thus sorted ascendingly by completion) and their associated prices. Then, iterate through the nodes and assign the cost to be the price of that node, given it is a sink (it has no children). This is because the smallest price reachable for a node with no children is itself. Then, if there are children, check the total cost for these children, if it is greater than or equal to the current cost of the node, do nothing, otherwise make the cost of the node equal to this cheaper cost.

Pseudo code:

```
cheapest _node (G=V,E):
    Create an empty array C to store the total cost for each node
    Run DFS(G), compute finish times and fill in array F of nodes in order of
     ascending finish times and array P the price of these nodes in the same
order.
    for u in 1 to V.length do
        C[u] = P[u]
        for v in edges of u do
            if C[v] < P[u]
                C[u]=C[v]
    return(C,F)
```

Correctness:

The reason this works is because by the time we calculate the minimum cost of a node, given it is ordered in ascending finish times, all of that node's children's cheapest costs have been calculated already. That way, if the current node could have reached a smaller price node further on in its path, the minimum cost associated with its direct child would be equal to that minimum cost because its minimum cost has already been calculated for all nodes further on in the path. This way, we can ensure there is not a node further on in the path that would have a smaller cost than the direct edges of the current node. Thus the loop variant is correct.

Base case:

For a single node, BFS returns just that 1 node, and then the price assigned to that node is its price. Thus, the base case is correct, the cheapest node reachable by the single node is itself.

Hypothesis:

All cheapest paths for all nodes are calculated

Loop Variant:

If a node further on in the path had a cheaper cost, it would have been found first because we are analyzing the output in ascending order of finish time. Thus, when we compare it to a node that finishes later, the minimum cost will already be associated with the closest node, thus the loop variant is correct.

Termination:

The algorithm terminates when all nodes have been discovered and all nodes cheapest costs have been found.

Running Time:

Because this is just standard DFS with the addition of another loop of O(n+m), the upper bound for this algorithm is O(n+m).

3. ***(35 points) You are given a string of n characters s[1, . . . , n], which you believe to be a corrupted text document in which all punctuation has vanished (e.g., "itwasthebestoftimes"). You wish to reconstruct the document using a dictionary available in the form of a Boolean function dict(·): for any string w, dict(w) = 1, if w is a valid word 0, otherwise Give a O(n 2)***

Kate Lassiter

dynamic programming algorithm that answers yes if the string s[·] can be reconstituted as a sequence of valid words, and no otherwise. You may map yes and no to 1 and 0, respectively. If the algorithm answers yes, you should also output the corresponding sequence of words. (Assume calls to dict take constant time.)

Algorithm description:

This algorithm hinges on the fact that once we find the optimal split for the start of the string from $s_i \ldots s_n$, we can then in one final iteration find the rest of the substrings within. It first starts by iterating through the string one letter at a time, and then if a valid word can be found between the first i letters, then the rest of the string is checked to see if a valid sequence of letters can be found. If it finds one, it initializes the index to keep searching from that point in the rest of the string. If it finds a meaningful divide on the last string, this means the string can be validly reconstructed as a sentence.

Pseudocode

```
stringer_fixer(s):
    Initialize array A of size n 0's to store indices where words start
    n=A.length
    result=0
    for i=1 to n do
        if dict[s[1:i]] == 1
            A[i]=1
            if s.length ==1 and A[i] == 1:
                result=1
                return(result,s)
            if i != n:
                start=i+1
                for j=i+1 to n:
                    if dict[s[start:j] == 1:
                        A[j]=1
                        start=j
                    if j==n and A[j]==1:
                        split_string=Split string where A = 1
                        result=1
                        return(result, split_string)
    return(result)
```

Subproblems/Recurrence:

The problem starts off with the first subproblem, finding the optimal string split for the first word, and then finding the optimal split for the rest of the substrings that follow. The equation can be derived as, where $i<j<k<n$:

$OPT(i,n) = OPT(1..i) \& OPT(i+1..j) \& OPT(j+1...k) \& OPT(k+1..n))$

The optimal split is on some letter i and then the rest of the string can have many optimal splits found for them up until n. Given you have solved OPT(1…i), the rest can be found.

Running Time:

Kate Lassiter

There is a nested for loop, both of size potentially n, so we have an $O(n^2)$ algorithm. All the other operations, initializing and if conditions are in constant O(1) time. Assume splitting the string into the final sequence takes constant time. Total running time is $O(n^2)$.
<u>Space:</u>
This algorithm stores just one array A, to indicate whether a given string index marks the end of a word or not, so it has an additional O(n) space complexity.