Solutions

Friday, May 7, 2021 1:02 PM

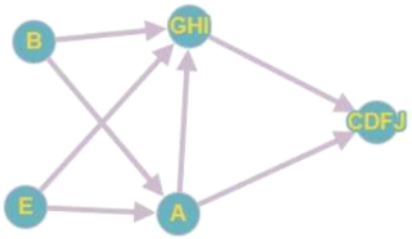
(a) $n - 100 = \Theta(n - 200)$
(b) $n^{1/2} = O(n^{2/3})$
(c) $100n + \log n = \Theta(n + (\log n)^2)$
(d) $n \log n = \Theta(10n \log 10n)$
(e) $\log 2n = \Theta(\log 3n)$
(f) $10 \log n = \Theta(\log(n^2))$
(g) $n^{1.01} = \Omega(\log^2 n)$
(m) $n2^n = O(3^n)$
(n) $2^n = \Theta(2^{n+1})$
(o) $n! = \Omega(2^n)$

3.4 i)

a) Order (multiple exists, but we want alphabetical) {C,D,F,J,G,H,I,A,B,E}

b) {E} and {B} are sources, and {CDFJ} is a sink

c)

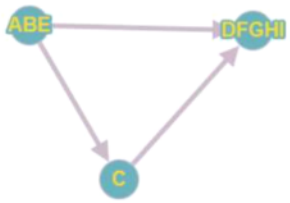


d) One edge must go from a vertex in {CDFJ} to either {E} or {B}. A second Edge must go from any vertex to the other source.

ii)

a) Order (again alphabetical) {D,F,G,H,I,C,A,B,E}

b) {ABE} is a source, and {DFGHI} is a sink



c)

d) One edge must go from any vertex in {DFGHI} to any vertex in {ABE}

3.8 (a) Let $G = (V, E)$ be our (directed) graph. We will model the set of nodes as triples of numbers (a_0, a_1, a_2) where the following relationships hold: Let $S_0 = 10$, $S_1 = 7$, $S_2 = 4$ be the sizes of the corresponding containers. a_i will correspond to the actual contents of the i^{th} container. It must hold $0 \leq a_i \leq S_i$ for $i = 0, 1, 2$ and at any given node $a_0 + a_1 + a_2 = 11$ (the total amount of water we started from). An edge between two nodes (a_0, a_1, a_2) and (b_0, b_1, b_2) exists if both the following are satisfied:

- the two nodes differ in exactly two coordinates (and the third one is the same in both).
- if i, j are the coordinates they differ in, then either $a_i = 0$ or $a_j = 0$ or $a_i = S_i$ or $a_j = S_j$.

The question that needs to be answered is whether there exists a path between the nodes $(0, 7, 4)$ and

$(*, 2, *)$ or $(*, *, 2)$ where $*$ stands for any (allowed) value of the corresponding coordinate.

(b) Given the above description, it is easy to see that a DFS (or BFS) algorithm on that graph should be applied, starting from node $(0, 7, 4)$ with an extra line of code that halts and answers 'YES' if one of the desired nodes is reached and 'NO' if all the connected component of the starting node is exhausted an no desired vertex is reached.

3.25 Start by linearizing the DAG. Let $v_1, \ldots v_n$ be the linearized order. Then the following algorithm finds the cost array in linear time.

```
find costs() {
    for i = n to 1: cost[v_i] = p_{v_i}
        for all (v_i, v_j) ∈ E:
        if cost[v_j] < cost[v_i]: cost[v_i] =
        cost[v_j]
}
```

The time for linearizing a DAG is linear. For the above procedure, we visit each edge at most once and hence the time is linear. For a general graph, the cost value of any two nodes in the same strongly connected component will be the same since both are reachable from each other. Hence, it is sufficient to run the above algorithm on the DAG of the strongly connected components of the graph. For a node corresponding to component C, we take $p_C = \min_{u \in C}\{p_u\}$.

False, consider the case where the heaviest edge is a bridge (is the only edge connecting two connected components of G.

True, consider removing e from the MST and adding another edge belonging to the same cycle. Then we get e new tree with less total weight.

True, consider the cut that has u in one side and v in the other, where $e = (u, v)$.

False,assume the graph consists of two adjacent 4-cycles,the one with very heavy edges of weight M and the other with very light edges of weight m. Let e be the edge in the middle with weight $m < w(e) < M$. Then the MST given by Kruskal, will pick all edges of weight m first and will not include e in the MST.

2.4. a) This is a case of the Master theorem with $a = 5, b = 2, d = 1$. As $a > b^d$, the running time is

$O(n^{\log_b a}) = O(n^{\log_2 5}) = O(n^{2.33})$.

b) $T(n) = 2T(n-1)+C$, for some constant C. $T(n)$ can then be expanded to $O(2^n)$.

c) This is a case of the Master theorem with $a = 9, b = 3, d = 2$. As $a = b^d$, the running time is

$O(n^d \log n) = O(n^2 \log n)$.

The running time of c) is much better than the others.

2.23. a) If A has a majority element v, v must also be a majority element of A_1 or A_2 or both. To find v, recursively compute the majority elements, if any, of A_1 and A_2 and check whether one of these is a majority element of A. If neither is a majority element, then you return that none exists. The running time is given by $T(n) = 2T(n/2) + O(n) = O(n \log n)$.

Note that just because an element is a majority element of A_1 or A_2 does not mean that it is also a majority element of A. This is why the algorithm must still check whether one of these is a majority element of A.

b) After this procedure described in the Hint, there are at most $n/2$ elements left as at least one element in each pair is discarded.

Claim: If A has a majority element, then it is still a majority element after this procedure.

Proof of Claim: Let k be the number of elements left after the procedure. If A has a majority element x, then it appeared in A strictly more than n/2 times. Suppose that x appears c times after the procedure.

Let us consider all the different elements that were in A. There were c pairs of x's, which resulted in c copies of x after the procedure. There were also k-c pairs of elements not equal to x, which resulted in the other k-c elements after the procedure. There were also at least n/2+1-2c instances of x, which were removed by pairing them with elements not equal to x. Finally, there must have been at least n/2+1-2c elements not equal to x to pair those with. In total, this gives us at least 2c+2(k-c)+2(n/2+1-2c) elements of A. This equals n+2k+2-4c, and yet we know this must be at most n, which is the number of elements in A. Thus, it must be that c > k/2, which means that x is still the majority element even after this procedure. This completes the proof.

Note that the converse of the Claim above is not true. For example, consider the pairs (1,1), (1,1), (2,3), (4,5), (6,7). There is no majority element originally, but after running the procedure, we get the set 1,1 that has a majority element. Therefore, the full algorithm must be as follows:

- Run the procedure described in the Hint
- If after the procedure there is no majority element, return "No majority"
- If after the procedure there is a majority element x, then check if x is a majority element of A in O(n) time. If it is, return x as the majority element, if it is not, return "No majority".

The running time of this algorithm is described by the recursion $T(n) = T(n/2) + O(n)$. Hence, $T(n) = O(n)$. Note that without the last "checking if x is really a majority element" step this algorithm would not be correct, since x could be a majority element after the procedure, but not be a majority element of A, as in the example above.

Longest Palindromic Subsequence: This can be solved using Dynamic Programming, since either the longest palindromic subsequence on A[i..j]:

Case 1: Does not contain A[i], and thus equals S[i+1][j], or

Case 2: Does not contain A[j], and thus equals S[i][j-1], or

Case 3: Contains both A[i] and A[j], and thus equals A[i] + S[i+1][j-1] + A[j]. This case is only possible if A[i]=A[j], since otherwise the sequence would not be palindromic.

But we don't know which of these cases is the correct one. However, since S[i][j] is the longest subsequence, we just take the "best" of these 3 cases (the one that generates the longest subsequence). So to compute S[i][j], we compute all of those 3 cases (the third case is allowed only if A[i]=A[j]), and then set S[i][j] to be the one with the longest result.

Note that to compute S[i][j], we need to first compute S[i+1][j] and S[i][j-1] and S[i+1][j-1]. So, we should start by computing S[i][j] for i and j being close to each other, and then compute S[i][j] for i and j being farther and farther apart.

To do this, we can start by computing S[i][i] (always equals A[i]) and S[i][i+1] (equals both A[i] and A[i+1] if A[i]=A[i+1], otherwise it is just either A[i] or A[i+1]). We do this for all i, these are the base cases. Afterwards, we can apply the recurrence above to compute S[i][j]; as long as we compute S[i+1][j] and S[i][j-1] and S[i+1][j-1] before S[i][j], any ordering will do. The final answer is S[1][n].

The longest palindromic subsequence in [7; 2; 4; 6; 9; 11; 2; 6; 10; 6; 15; 6; 14; 2; 7; 5; 13; 9; 12; 15] is [7, 2, 6, 6, 10, 6, 6, 2, 7]. In a sequence of 1000 random numbers from 1 to 100, the size of the longest palindromic subsequence should be approximately 170 to 180.

Longest common substring:
Base Case: If string is empty LCS will be 0.

Recurrence:

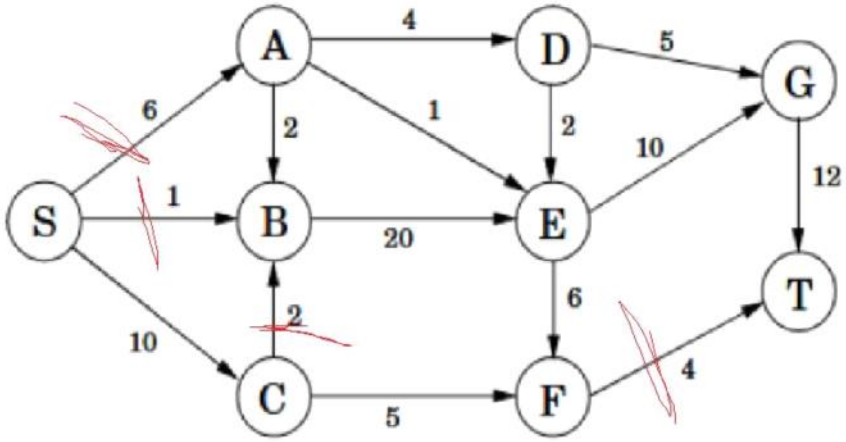
Check if the ith character in first string is equal to the jth character the second string. If they are, then the length is equal to the length of moving one character back plus one. LCS[i][j] = LCS[i-1][j-1] + 1. If the character is different, there is no commonality and LCS[i][j] = 0

Iterate for all i and for all j starting at 0,0 and working up to [n,m]. Once you have filled the table select the i,j pair with highest value. This pair gives the end location of the substring, and the value in the table gives its length (which can be used to find the start of the string.

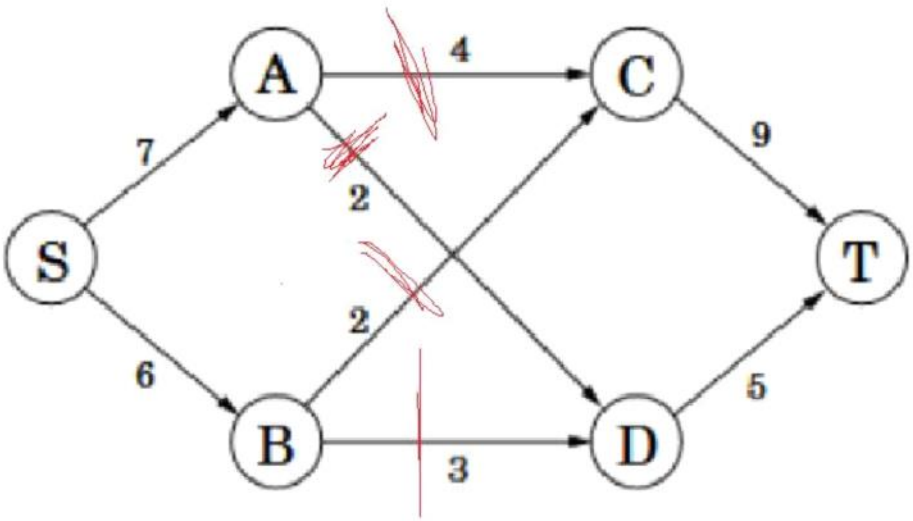
O(nm)

(7.2) Let f_{KN} be the amount transported from Kansas to Mexico, and similarly for f_{KC}, f_{MN}, and f_{MC}. The objective is to minimize $4f_{MN} + f_{MC} + 2f_{KN} + 3f_{KC}$. The constraints are that $f_{KC} + f_{KN} \leq 15$, $f_{MC} + f_{MN} \leq 8$, $f_{KN} + f_{MN} \geq 10$, and $f_{KC} + f_{MC} \geq 13$.

Max flow is 12
Min cut is seen below:



Max flow is 11
Min cut is seen below



8.9 This is a generalization of VERTEX-COVER. Given a graph G, consider each edge e = (u, v) as a set containing the elements u and v. Then, finding a hitting set of size at most b in this particular family of sets is the same as finding a vertex cover of size at most b for the given graph. The problem is in NP, since given a valid solution, we can verify it is correct in polynomial time.

8.13 (a) This can be solved in polynomial time. Delete all the vertices in the set L from the given graph and find a spanning tree of the remaining graph. Now, for each vertex $l \in L$, connect it to any of its neighbors present in the tree. It is clear that such a tree, if it is possible to construct one, must have all the vertices in L as leaves. If the graph becomes unconnected after removing L, or some vertex in L has no neighbors in G\L, then no spanning tree exists having all vertices in L as leaves.

(b) This generalizes the (undirected) (s, t)-RUDRATA PATH problem. Given a graph G and two vertices s and t, we set L = {s, t}. We now claim that the tree must be a path between s and t. It cannot branch out anywhere, because each branch must end at a leaf and there are no other leaves available. Also, since it is a spanning tree, the path must include all the vertices of the graph and hence must be Rudrata path. Similarly, every Rudrata path is a tree of the type required above. The problem is also in NP, since given a valid solution, we can verify it is correct in polynomial time.

(c) This is also a generalization of (undirected) (s, t)-RUDRATA PATH. We use the same reduction as in the previous part. Note that a tree must have at least two leaves. Hence for L = {s, t}, the set of leaves must be exactly equal to L. The problem is also in NP for the same reason.