CS624: Analysis of Algorithms

Assignment 4

Due: Saturday, April 11, 2026

1. Determine an LCS of $< 1, 0, 0, 1, 0, 1, 0, 1 >$ and $< 0, 1, 0, 1, 1, 0, 1, 1, 0 >$. Of course you could probably solve this problem easily enough just by looking at it. What I want you to do is to explicitly use the algorithm presented in class. Write out your derivation neatly, including the table.

   Here is a table with the path marked. Notice that more than one solution exists. If both left and up arrow were equivalent I chose up, but other paths are possible.

	y_j	1	0	0	1	0	1	0	1
x_i	0	0	0	0	0	0	0	0	0
0	0	↑0	↖1	↖1	↑0	↖1	↑0	↖1	↑0
1	0	↖1	↑1	↑1	↖2	↑1	↖2	↑1	↖2
0	0	↑1	↖2	↖2	↑2	↖3	←3	↖3	←3
1	0	↖1	↑2	↑2	↖3	↑3	↖4	←4	↖4
1	0	↖1	↑2	↑2	↖3	↑3	↖4	↑4	↖5
0	0	↑1	↖2	↖3	↑3	↖4	↑4	↖5	↑5
1	0	↖1	↑2	↑3	↖4	↑4	↖5	↑5	↖6
1	0	↖1	↑2	↑3	↖4	↑4	↖5	↑5	↖6
0	0	↑1	↖2	↖3	↑4	↖5	↑5	↖6	↑6

   Notice that there are many possible solutions, all of size 6

2. Suppose you are managing the construction of billboards on the highway, a heavily-traveled stretch of road that runs west-east for M miles. The possible sites for billboards are given by numbers $x_1, x_2, \ldots, x_n$, each in the interval $[0, M]$ (specifying their position along the highway, measured in miles from its western end). If you place a billboard at location x_i , you receive a revenue of $r_i > 0$. You want to place billboards at a subset of the sites in $x_1, \ldots, x_n$ so as to maximize your total revenue, subject to the following restrictions:

   - You cannot build two billboards within 5 miles or less of one another on the highway.
   - You cannot build a billboard within 5 miles or less of the western or eastern ends of the highway.

   A subset of sites satisfying these two restrictions will be called valid. For example, Suppose M = 20, n = 4, $x_1, x_2, x_3, x_4 = 6, 7, 12, 14$, and $r_1, r_2, r_3, r_4 = 5, 6, 5, 1$. Then the optimal solution would be to place billboards at x_1 and x_3 with a revenue of 10.

   (a) Describe a recursive algorithm that solves the problem (inefficiently). Hint: You either select x_n and collect the revenue or not... What is the solution in each case?

Answer: As hinted, you either select x_n or you don't. If you do, you collect the revenue and recursively solve the problem for $x_1 \dots x_i$, where x_i is the latest index you can still use – that is, within more than 5 miles of x_n. If you don't select x_n, you solve recursively for $x_1 \dots x_{n-1}$.

(b) Show that the problem has optimal substructure and overlapping subproblems, and formulate the dynamic programming approach.

Answer: The optimal substructure can be shown by a cut and paste argument. The structure of the optimal solution is a sequence of billboards. If we take a subsequence $x_i \dots x_j$ covering part of the road, it is an optimal selection for this part. If we could find a better arrangement that turns a better revenue we could place it and have a better solution overall, contradicting the optimality of the original solution.

There are also overlapping subproblems, because we have to potentially calculate segments over and over again as we recurse over them.

(c) Describe the dynamic programming solution, including the array(s) for the example above.

Answer: We can use the following formula to describe bill(i), as the optimal revenue for miles 1 to i over the set $x_1 \dots x_i$:

$$bill(i) = \begin{cases} 0 & \text{if } i <= 0 \\ \max\{r_i + bill(j), bill(i-1)\} & \text{otherwise} \end{cases}$$

Where bill(j) represents the closest billboard that is more than 5 miles west of x_i In this case bill(6) = 5, bill(7) = 6, bill(12) = 10 and bill(14) = 10.

3. Professor Stewart is consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure: that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.

   Professor Stewart is given the tree that describes the structure of the corporation, using the left-child, right-sibling representation (as in – every node has a pointer to its leftmost child and to its next sibling on the right in a singly linked list). Each node of the tree holds, in addition to the pointers, the name of an employee and that employee's conviviality ranking. Describe a dynamic programming algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.

   Solution: For every node v, you calculate the maximum conviviality score of the subtree rooted at that node denoted $C(v)$ by calculating the maximum of two scores, I(v) for invite or E(v) for exclude. There are two options: If you invite the employee represented by that node you add his/her conviviality score (denoted c(v)) but you have to exclude all of his/her direct underlings. If you don't invite that employee, you calculate the sum of the maximum scores of his/her underlings and take the maximum over the two. The boundary condition is a leaf – an employee without any underlings, where I(v)= c(v) and E(v) = 0. For every other node, $I(v) = \max\{c(v) + \sum_{u \in Children(v)} E(u)\}$, and $E(u) = \sum_{u \in Children(v)} \max\{I(u), E(u)\}$. We can calculate it bottom up, and go over every node once per calculation- so it's a linear time algorithm.

4. The palindrome problem is a bit like the LCS problem, in the sense that you have to compare the first and last character of the string – $A[1]$ and $A[n]$. If they are the same, then they are part of a palindrome, and so you can recursively find the longest palindrome spanning $A[2..n-1]$ and add $A[1]$ and $A[n]$. If they are not the same, at least one of them should be discarded, so the longest palindrome is either the longest of $A[2..n]$ or of $A[1..n-1]$. Calculate

both and get the maximum. Boundary: If A's size is 0 or 1, it's trivially a palindrome. The formula is as follows:

$$PAL(i,j) = \begin{cases} 0 & \text{if } j < i \\ 1 & \text{if } j = i \\ 2 + PAL(i+1, j-1) & \text{if } A[i] = A[j] \\ \max\{PAL(i+i, j), PAL(i, j-1)\} & \text{otherwise} \end{cases}$$

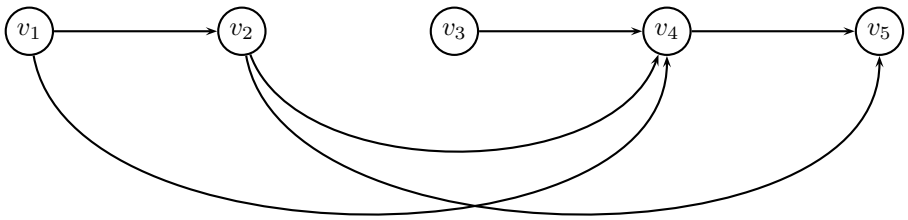
A DP algorithm calculates $PAL[1, n]$ by memoizing all the values of $PAL[i, j]$ for $i \leq j$. This is a quadratic algorithm.

(you can also calculate the LCA of the string and its reverse).

5. Let $G = (V, E)$ be a directed graph with vertices $V = \{v_1, v_2, \ldots, v_n\}$. The set E of edges has the following property:

   - Each edge goes from a vertex of lower index to a vertex of higher index. For instance, there might be an edge (v_1, v_7) from v_1 to v_7. But there definitely will *not* be an edge (v_7, v_1) from v_7 to v_1.
   - Each vertex except v_n has at least one edge leaving it. That is, for every vertex v_i $(1 \leq i \leq n-1)$, there is at least one edge of the form (v_i, v_j) (of course with $i < j$).

   Here is an example of a graph with this property:



   It is called a DAG (directed acyclic graph). We will discuss them later on in the course. By a *path* we mean as usual a sequence of vertices $\{v_{i_1}, v_{i_2}, \ldots v_{i_k}\}$,

   such that for each successive pair in the sequence there is an edge from the first to the second. (That is, there is an edge from v_{i_1} to v_{i_2}, another edge from v_{i_2} to v_{i_3}, and so on.)

   The *length* of a path is just the number of edges in it. Equivalently, it is 1 less than the number of vertices in it.

   We want to find an algorithm to solve the following problem, given such a graph:

   > Find the length of the longest path that begins at v_1 and ends at v_n.

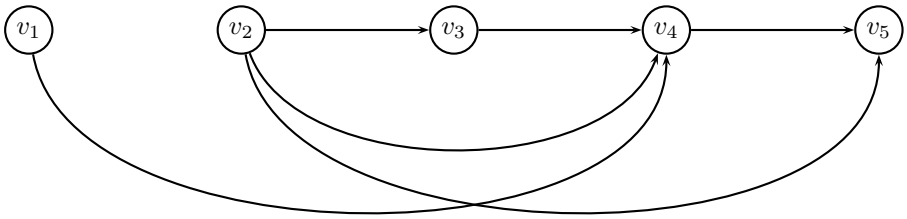
   Here is a "greedy" algorithm that it is natural to think of at first:

Algorithm 1 FindPath

```
1: w ← v_1  w is the current node we are considering.
2: L ← 0  L will hold the greatest length of any path so far.
3: while there is an edge out of w do
4:    Choose the edge (w, v_j) for which j is as small as possible.
5:    w ← v_j
6:    L ← L + 1
7: end while
8: return L
```

   (a) Show that this algorithm gives the correct answer for the graph drawn. The path returned by the algorithm is $v_1 \to v_2 \to v_4 \to v_5$, which is indeed the longest.

b. Slightly modify the graph in (a) such that this algorithm no longer gives the longest path (**Hint:** It's enough to modify two edges: Delete (v_1, v_2) and add another edge, you have to find out where). Explain briefly why the algorithm above won't work on your example.



The algorithm in (a) will give $v_1 \to v_4 \to v_5$, while the longest path is $v_2 \to v_3 \to v_4 \to v_5$. Other examples exist too. I accepted any example that works.

c. It is possible to find the longest path using dynamic programming. This is done by calculating, for every vertex v_i, the longest path that ends in v_i. Notice that any vertex v_i extends the longest paths ending at each one of its incoming neighbors by 1 (this is true only in DAGs, of course, due to the fact that paths only go one way, so to speak). Therefore, the length of the longest path ending at v_i is the length of the longest path of all of v_i's predecessors + 1. The algorithm is given below:

i. For each v_i whose in-degree is 0, set $LP(v_i) = 0$ (this is the longest path ending at v_i).

ii. For each of the other vertices w_i, in the topological sort order:

A. set $LP(w_i) = \max_{v_j s.t (v_j, v_i) \in E} LP(v_j) + 1$

iii. Return $\max_{v_i \in V}(LP(v_i))$

For each of the vertices in the DAG shown in (a) above, fill in the length of the longest path ending in it.

v_i	$LP(v_i)$
v_1	0
v_2	1
v_3	0
v_4	2
v_5	3

d. (4%) What is the run time of the algorithm in (c) above as a function of $|V|$ (the number of vertices) and/or $|E|$ (the number of edges)? Explain briefly.

$O(|V| + |E|)$ Notice that the inner loop in the algorithm above depends solely on the number of edges. Hence, the run time is $O(|V| + |E|)$.

6. MST:

(a) Given a graph $G = (V, E)$ and MST, if we look at any subgraph $G' = (V', E')$, the portion of the MST that spans G' is optimal with respect to G'. The proof is a simple cut+paste argument - if this were not the case, we could have replaced the set of edges with a cheaper way to span the subgraph, creating an overall better MST, contradicting the initial assumption of a minimum spanning tree.

(b) First of all, for each cut there must be an MST edge that crosses it, or the tree would be disconnected. Say, by contradiction that for a certain cut, the MST edge that crosses it, e, is not the lightest. Then we can remove e, disconnect the tree and reconnect it using e', the lightest edge that crosses the cut. We can then get a better tree, contradicting the assumption that the original MST is indeed the minimum one.

(c) This is a simple extension of (b) above. The lightest edge in the tree, $e = (u, v)$ must cross some cut - let's say u on one side, v on the other, and all other vertices anywhere we want.

7. Independent set:

(a) First notice that there is a bit of a twist here – if we select a node for a maximum independent set, we cannot select any of its children, so the substructure is all the subtrees rooted in the grandchildren. Therefore the optimal substructure is: Given a maximum independent set, then for every subtree (rooted in the grandchildren of selected nodes, not children), the part of the set has to be optimal for that subtree. The proof is a cut and paste.

(b) Again – be careful here. Not every leaf is a part of every independent set, so don't go that route. Given a maximum independent set S. Given any leaf l – if $l \in S$ we are good. Otherwise, its parent is in the set (if neither the leaf nor its parent are in the set it can't be maximum... think why). In this case we can remove the parent and add l. The new set S' is still independent, because the only neighbor of l is the parent that's now removed, and it is the same size, so still maximum.

(c) The recursive formula, given a tree T, is as follows:

$$MS(T) = \begin{cases} Null & \text{if T is empty} \\ T & \text{if T is a leaf} \\ \max\{root(T) + \cup MS(\text{T's grandchildren}), \cup MS(\text{T's children})\} & \text{otherwise} \end{cases}$$

In other words - if we pick a root, we recursively calculate the max. set of its grandchildren. Otherwise, we calculate the max set of its children, pick the best result.

However, we said in (b) that every leaf can be a part of a max. set, so we can do it greedily as follows: Pick a leaf at random, add to the set. Delete its parent and all the edges from/to it from consideration (the tree is then disconnedted but it's ok). Continue until done.