Evan Edelstein

EN.605.621.84.FA25

HW 3

The work in this exercise is mine alone without un-cited help. No AI was used to answer these questions.

1. Pseudocode:

Function AVL\_in\_range(a, b, node=T.root):

If node is NULL:

Return False

If a <= node.specific\_weight <= b:

Return True

Else if node. specific weight < a:

Return AVL\_in\_range(a, b, node.right)

Else if node. specific weight > b

Return AVL\_in\_range(a, b, node.right):

This algorithm is very similar to searching a binary search tree. The function AVL\_in\_range accepts three arguments, *a*,*b* and node, The first argument *a*, is the lower bound of the search range and the second parameter *b*, is the upped bound of the search range. The last parameter node is a node of an AVL tree T, that the search starts from. Each node in the AVL tree is assumed to have 2 pointers, left and right to the child nodes and a data attribute specific\_weight denoting the specific weight we are searching for. The function first is called with node equal to the root of T, represented by T.root. The algorithm then recursively searches the tree for a node within a specific weight in the range [a,b] inclusive.

If the node is Null, we have traversed all possible nodes with specific\_weight in range [a,b] without finding a hit so returns false. If the node is not null and its specific weight is in the range [*a*,*b]* (inclusive), it is a match, so we return true. If the nodes specific weight is less than *a,* we continue searching in the right subtree that contains values greater than *a*, as this follows from the BST property. Conversely, if the nodes specific weight is larger than *b* we search for smaller values in the left subtree. Since each recursive call travels one level of the tree and performs constant time operations for each recursive call The runtime of this algorithm is O(height of tree), which in the case is O(logn), by the BST property,

1. We can use optimal subproblems to define the bellman equation for this question. The minimum cost for week W, is the sum of the cost for all weeks up to W and the minimum cost of week W. We can use a dynamic programing approach to build up an array DP that stores the running tally of the minimum cost for each week. In the first 3 weeks the only option is to choose company A. That is by week three the minimum cost must be:

W(3) = S[0] \* r + S[1] \* r + S[2] \* r

Afterwards the cost for the week is either company A’s cost plus the previous weeks cost or Company B’s cost plus the cost for four weeks. Since we are building up subproblems from the bottom up we can “go back in time” to choose option B if the tally after 4 weeks is less than the cost of choosing B 4 weeks ago. Writing out

the bellman equation for a given week i:

DP[i] = {

if i = 0 Case A

if 0 < i < 3 Case B

if i >= 3 Case C

}

Case A refers to the first week, which does not need to lookup a previous result in DP, so is computed and stored in the first spot in DP. Case B is for the next 3 weeks, which adds weeks ‘s cost of using A to the tally of the cost (DP[i-1]). Case C, chooses the minimum of Case B or using company B for the last 4 weeks, using the look-behind method discussed above. We take the tally from four weeks before in the DP[1-4] lookup. If company B is chosen, this bellman equation properly updates the tally in DP[i], used for the next weeks, to have the correct cost tally.

1. (im using S instead of y for clarity) The string segmentation with the maximum quality can approached like the rod cutting example in the textbook. We can use dynamic programing to solve the subproblem of finding the highest quality substrings of the input, noting that the highest quality substring of a string S of length n, can be viewed as finding the highest quality prefixes of S, and then removing the prefix and recursing. For example, for the string “meetateight”, the highest quality prefix is “meet”. We then recursively call our algorithm on “ateight” to get the highest prefix “at”. Instead of using recursion we can build up the prefix quality using bottom-up approach using dynamic programing, because the subproblems of maximum prefixes exhibits independent substructure. For each position in S, denoted i, we find the highest quality cut of S at position j in S[0:i] to get a prefix from 0 to j and suffix from j to i, and store that sum of the prefix and suffix quality in the DP array. Since we use the DP array, we do not need to reevaluate the quality of the prefix S[0,j] each time it is needed, instead we can look it up in DP[j]. The Bellman equation for this process can be expressed as:

DP[i] = {

0 if i = 0

if i > 0

}

The best cut position j for each i in Scan be stored in an array and traversed backwards starting at n to get the proper segmentations of S.

The pseudocode for this algorithm would be:

Define segment\_str(S)

N = S.length

DP , cuts = array of length n

For i=1 to n+1:

max\_qual = -INF

for j=0 to i:

qual = DP[j] + quality(S[j:i])

if qual > max\_qual:

max\_qual = qual

cuts[i] = j

DP[i] = max\_qual

words = []

While n > 0:

c = cuts[n]

words.insert\_front(S[c:n])

n = c

return join(words)

The nested for-loop and set bounds (no breaks or conditional execution) of the loops indicates that this program has runtime of ). This algorithm can be proven using proof by contradiction based on the optimal and independent subproblems. Assume there exists some more optimal independent subproblem prefix segmentation P for some i,j in S where both I and j are less than S.length. This would mean we would have some i, for which P + Quality(S[j:i)) is greater than DP[j] + Quality(S[j:i), thus contradicting our bellman equation that selects the max DP[j] +Quality(S[j:i) for i. This we know that the bellman equation above finds the optimal prefix segmentation for S.

1. a) let K = 5 and have a set of containers of weights [5,4,3,2,1]. The greedy algorithm would use 4 trucks:

truck: containers

1: 5

2: 4

3: 3 + 2

4: 1

The optimal truck usage would be to stack container of weights 4 and 1 onto the same truck and use 3 trucks instead.

b) Let the minimum number of trucks for n containers using a greedy algorithm be denoted *G*. Let *O* be the minimum number of trucks needed using an optimal algorithm. We can show that *G* is at most double *O*, noting that as we pack a truck in the greedy algorithm there is some container that exceeds the weight of the truck and contributes to another truck being needed. At worst, these leftover containers contribute to half the total trucks (one new truck for each overflow container). The optimal algorithm would pack the trucks in a way that minimizes or even removes the trucks needed for the overflow containers. Thus, *G*/2 <= *O* or *G* <= 2*O*.