1 Tree Traversal

Here is one recursive solution to find the minimum fullness of any sub-tree of a given tree.

function findMinFullness(tree):
    """findMinFullness: tree -> double
    Purpose: Given a tree, find the minimum fullness of its subtrees
    """
    if tree is empty:
        throw empty tree exception
    return minFullnessHelper(tree.root())

function minFullnessHelper(node):
    """minFullnessHelper: node -> double
    Purpose: Find the minimum fullness of a tree whose root is node
    """
    node.height = 0
    node.size = 1
    minFullness = infinity
    if node.hasLeft():
        minFullness = min(minFullnessHelper(node.left), minFullness)
        node.height = max(height, (node.left).height + 1)
        node.size += node.left.size()
    if node.hasRight():
        minFullness = min(minFullnessHelper(node.right), minFullness)
        node.height = max(height, (node.right).height + 1)
        node.size += node.right.size()
    fullSize = 2 ^ (node.height + 1) - 1
    return min(minFullness, (node.size) / fullSize)

2 Pseudocode

Suppose you have a stack and you want to know whether it’s “almost empty”. Write pseudocode for a method that returns “true” if the stack is either empty or has just one item in it. The Stack class you’re enhancing does not keep track of its current size, n, but has an isEmpty() function. Your method should be $O(1)$, i.e., constant time.

def nearly_empty(stack):
    """nearly_empty: stack -> boolean
    Purpose: returns true if stack has 0 or 1 elements, otherwise false
    """
    if stack.isEmpty():  // n == 0
return true
popped = stack.pop()
isNearlyEmpty = false
if stack.isEmpty(): // n == 1
    isNearlyEmpty = true
stack.push(popped)
return isNearlyEmpty

3 Master Theorem

What is the big-O running time of a recursive algorithm that splits a problem of size $n$ into 3 subproblems each of size $2n/3$, recursively solves the three subproblems, and then combines the solutions in time $O(n^2)$?

$$T(n) = 3T(2n/3) + \Theta(n^2)$$

$a = 3, b = 3^2, d = 2$

$$b^d = \frac{9}{4}$$

$a > b^d$

$$O(n^{\log_3 3})$$

4 Sorting

You are given two arrays of integers, $A$ and $B$, both sorted in increasing order. $A$ has enough empty space at the end to include all the elements of $B$ within it. Write pseudocode for a method to merge $B$ into $A$ such that all the elements in the merged array $A$ are sorted in increasing order as well.

Example: $A = [1,3,5,7...]$ and $B = [2,4,6]$ then your method should return the array $[1,2,3,4,5,6,7]$.

Solution: This can be accomplished by using merge sort. Look at the merge sort pseudocode for help!

Here’s the pseudocode solution:

def merge(A,B):
    m = A.size
    n = B.size
    assert len(A) == m + n // Make sure that there’s space in A to hold B
    k = m + n - 1 // Index of end of A. Alternatively could be len(A) - 1
    i = m - 1 // Index of last non-None element in A
    j = n - 1 // Index of last non-None element in B
    while(i >= 0 and j >= 0):
        if(a[i] > b[j]):
            a[k] = a[i]
            i -= 1
        else:
            a[k] = b[j]
            j -= 1
        k -= 1
    while(j >= 0):
        a[k] = b[j]
        k -= 1
        j -= 1
Try hand simulating with your own examples!

5 Dynamic Programming

Before describing the solution we define some notation. A rope of length $x$ is denoted $\lfloor x \rfloor$. Furthermore, $\text{maxprice}(\lfloor x \rfloor)$ is the maximum price we can get from a rope of length $x$ (i.e., the price of the cut that maximizes the price) and $\text{price}(\lfloor x \rfloor)$ is the price of a rope of length $x$.

The first step is to find the recursive structure of the problem (i.e., the “magic step”). Notice that given a rope of length $n$, the maximum price we can get for it is the maximum between the following values:

- what we can get for a piece of length 1 plus the maximum we can get for a piece of length $n - 1$;
- what we can get for a piece of length 2 plus the maximum we can get for a piece of length $n - 2$;
- ...
- what we can get for a piece of length $n$ plus the maximum we can get for a piece of length 0.

More formally, we write this as the following recursive function:

$$\text{maxprice}(\lfloor n \rfloor) = \max \left\{ \text{price}(\lfloor 1 \rfloor) + \text{maxprice}(\lfloor n - 1 \rfloor), \right.$$  
$$\text{price}(\lfloor 2 \rfloor) + \text{maxprice}(\lfloor n - 2 \rfloor), \right.$$  
$$\ldots,$$
$$\text{price}(\lfloor n - 1 \rfloor) + \text{maxprice}(\lfloor 1 \rfloor),$$  
$$\text{price}(\lfloor n \rfloor) + \text{maxprice}(\lfloor 0 \rfloor) \right\}$$

To gain some intuition let’s consider the recursion tree of this function. Figure ?? gives the recursion tree of $\text{maxprice}(\lfloor 4 \rfloor)$. Note that for visual clarity we write $\text{maxprice}(x)$ instead of $\text{maxprice}(\lfloor x \rfloor)$ in Figure ???. We can see from the recursion tree that the sub-problems are overlapping; that is, the sub-problems need to be solved for more than one problem. For example, $\text{maxprice}(\lfloor 1 \rfloor)$ is a sub-problem of $\text{maxprice}(\lfloor 2 \rfloor)$, $\text{maxprice}(\lfloor 3 \rfloor)$ and $\text{maxprice}(\lfloor 4 \rfloor)$. Similarly, $\text{maxprice}(\lfloor 1 \rfloor)$ is a sub-problem of $\text{maxprice}(\lfloor 2 \rfloor)$, $\text{maxprice}(\lfloor 3 \rfloor)$ and $\text{maxprice}(\lfloor 4 \rfloor)$. So using Dynamic Programming for this problem makes sense!

Next, we have to figure out in what order to solve the sub-problems. Looking at the recursion tree it is clear that we should start with $\text{maxprice}(\lfloor 0 \rfloor)$, then $\text{maxprice}(\lfloor 1 \rfloor)$, then $\text{maxprice}(\lfloor 2 \rfloor)$, etc. This way, we will have the solution of $\text{maxprice}(\lfloor 0 \rfloor)$ ready when we need to solve $\text{maxprice}(\lfloor 1 \rfloor)$, and we will have the solution of $\text{maxprice}(\lfloor 1 \rfloor)$ ready when we need to solve $\text{maxprice}(\lfloor 2 \rfloor)$ and so on and so forth.

Now that we know in what order we need to solve the sub-problems, we have to design an iterative algorithm that will solve the sub-problems and store the solutions somewhere so we can re-use them. At a high-level, we will use an array $T$ where $T[i]$ will store $\text{maxprice}(\lfloor i \rfloor)$. If we write this out explicitly we have:

$$T[0] = \text{maxprice}(\lfloor 0 \rfloor) = 0$$  
$$T[1] = \text{maxprice}(\lfloor 1 \rfloor) = \max \left\{ \text{price}(\lfloor 1 \rfloor) + \text{maxprice}(\lfloor 1 - 1 \rfloor) \right\}$$  
$$T[2] = \text{maxprice}(\lfloor 2 \rfloor) = \max \left\{ \text{price}(\lfloor 2 \rfloor) + \text{maxprice}(\lfloor 2 - 1 \rfloor), \text{price}(\lfloor 2 \rfloor) + \text{maxprice}(\lfloor 2 - 2 \rfloor) \right\}$$  
$$\ldots$$  
$$T[n] = \ldots$$
Figure 1: Recursion tree of \texttt{maxprice(4)}. 

We now describe the algorithm detail:

```python
def ropeCut(prices, length):
    """
    Consumes: prices \rightarrow list of prices for each length of rope
    length \rightarrow length of the rope we have
    Produces: integer
    Purpose: Determines the maximum profit possible from our given rope
    """
    # initialize array of length + 1 size
    \[ \text{int}\[\]\ T = \text{new int}[\text{length} + 1] \]
    for i from 0 to length of T:
        # initialize each element to 0
        T[i] = 0
    for i from 1 to length:
        for j from 1 to i:
            T[i] = max(T[i], prices[j - 1] + T[i - j])
    # return the best profit for our rope
    return T[length]
```

The time complexity of this solution is $O(n^2)$.

If you're not too confident about dynamic programming, be sure to look back at HW3 for some more review! You can also google Dynamic Programming practice problems to find some problems for practice.