CS16 Practice Midterm 2021 Solutions

due Never

1 Dynamic Programming

Before describing the solution we define some notation. A rope of length x is denoted $\overset{x}{\vdash\!\!\dashv}$. Furthermore, $\mathsf{maxprice}(\overset{n}{\vdash\!\!\dashv})$ 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 $\mathsf{price}(\overset{x}{\vdash\!\!\dashv})$ 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 apiece of length $n-2$;
- ...
- what we can get for a piece of length n plus the maximum we can get for a apiece of length 0.

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

$$
\begin{aligned}
\mathsf{maxprice}(\overset{n}{\vdash\!\!\dashv}) = \max \Big\{ & \mathsf{price}\left(\overset{1}{\vdash\!\!\dashv}\right) + \mathsf{maxprice}\left(\overset{n-1}{\vdash\!\!\dashv}\right), \\
& \mathsf{price}\left(\overset{2}{\vdash\!\!\dashv}\right) + \mathsf{maxprice}\left(\overset{n-2}{\vdash\!\!\dashv}\right), \\
& \dots, \\
& \mathsf{price}\left(\overset{n-1}{\vdash\!\!\dashv}\right) + \mathsf{maxprice}\left(\overset{1}{\vdash\!\!\dashv}\right), \\
& \mathsf{price}\left(\overset{n}{\vdash\!\!\dashv}\right) + \mathsf{maxprice}\left(\overset{0}{\vdash\!\!\dashv}\right) \Big\}
\end{aligned}
$$

To gain some intuition let's consider the recursion tree of this function. Figure 1 gives the recursion tree of $\mathsf{maxprice}(\overset{4}{\vdash\!\!\dashv})$. Note that for visual clarity we write $\mathsf{maxprice}(x)$ instead of $\mathsf{maxprice}(\overset{x}{\vdash\!\!\dashv})$ in Figure 1. 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, $\mathsf{maxprice}(\overset{0}{\vdash\!\!\dashv})$ is a sub-problem of $\mathsf{maxprice}(\overset{1}{\vdash\!\!\dashv})$, $\mathsf{maxprice}(\overset{2}{\vdash\!\!\dashv})$, $\mathsf{maxprice}(\overset{3}{\vdash\!\!\dashv})$ and $\mathsf{maxprice}(\overset{4}{\vdash\!\!\dashv})$. Similarly, $\mathsf{maxprice}(\overset{1}{\vdash\!\!\dashv})$ is a sub-problem of $\mathsf{maxprice}(\overset{2}{\vdash\!\!\dashv})$, $\mathsf{maxprice}(\overset{3}{\vdash\!\!\dashv})$ and $\mathsf{maxprice}(\overset{4}{\vdash\!\!\dashv})$. 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 $\mathsf{maxprice}(\overset{0}{\vdash\!\!\dashv})$, then $\mathsf{maxprice}(\overset{1}{\vdash\!\!\dashv})$, then $\mathsf{maxprice}(\overset{2}{\vdash\!\!\dashv})$, etc. This way, we will have the solution of $\mathsf{maxprice}(\overset{0}{\vdash\!\!\dashv})$ ready when we need to solve $\mathsf{maxprice}(\overset{1}{\vdash\!\!\dashv})$, and we will have the solution of $\mathsf{maxprice}(\overset{1}{\vdash\!\!\dashv})$ ready when we need to solve $\mathsf{maxprice}(\overset{2}{\vdash\!\!\dashv})$ and so on an 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

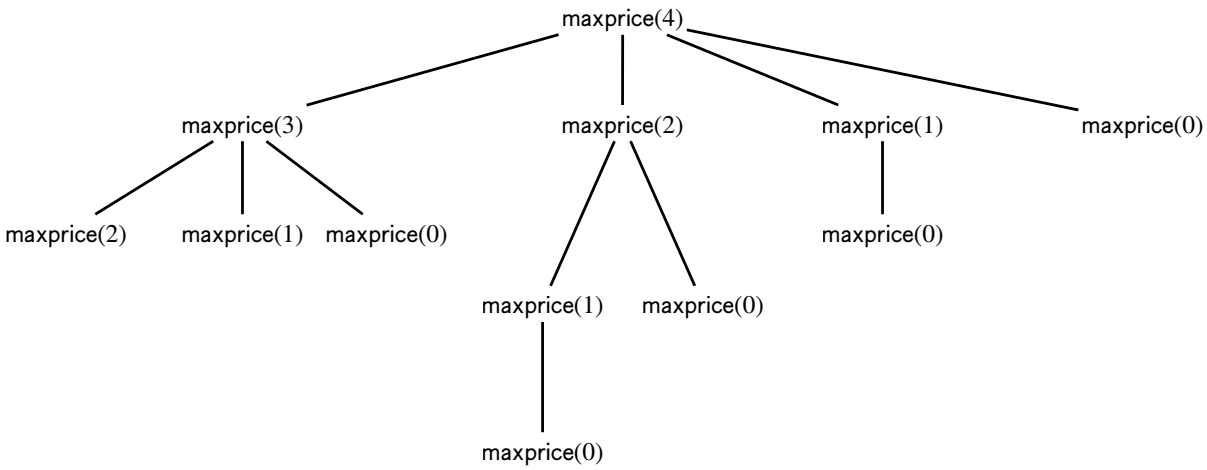


Figure 1: Recursion tree of maxprice(4).

a high-level, we will use an array T where T[i] will store maxprice($\vdash\!\overset{i}{\!\dashv}$). If we write this out explicitly we have:

$$\begin{aligned}
\texttt{T}[0] &= \mathsf{maxprice}(\vdash\!\overset{0}{\!\dashv}) = 0 \\
\texttt{T}[1] &= \mathsf{maxprice}(\vdash\!\overset{1}{\!\dashv}) = \max\left\{\mathsf{price}(\vdash\!\overset{1}{\!\dashv}) + \mathsf{maxprice}(\vdash\!\overset{1-1}{\!\dashv})\right\} \\
\texttt{T}[2] &= \mathsf{maxprice}(\vdash\!\overset{2}{\!\dashv}) = \max\left\{\mathsf{price}(\vdash\!\overset{1}{\!\dashv}) + \mathsf{maxprice}(\vdash\!\overset{2-1}{\!\dashv}), \mathsf{price}(\vdash\!\overset{2}{\!\dashv}) + \mathsf{maxprice}(\vdash\!\overset{2-2}{\!\dashv})\right\} \\
&\ldots \\
\texttt{T}[n] &= \ldots
\end{aligned}$$

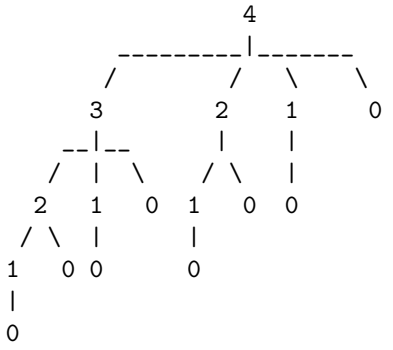
A naive solution of this problem could be to partition the rope of length n into two parts of length i and $n - i$. We recurse only for rod length $n - i$. Finally we take the maximum of all values.

```
def ropeCut(prices, length):
    """
    Consumes: prices -> list of prices for each length of rope
              length -> length of the rope we have
    Produces: integer
    Purpose: Determines the maximum profit possible from our given rope
    """

    # base case
    if length == 0:
        return 0

    int maxValue = negative infinity

    for i from 1 to length:
        # recurse for length - i remaining rope
        int cost = prices[i - 1] + ropeCut(prices, length - i)

        # find maximum profit
        if cost > maxValue:
            maxValue = cost
```

```
    return maxValue
```

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

This problem can be broken down into smaller subproblems that can further be broken down into even smaller subproblems. For a rope of length 4, we end up with a recursion tree like this:



As you can see, the same subproblems are computed multiple times. Knowing that, we can solve this problem starting at the bottom of the tree, solve the smaller subproblems first, and work our way up from there. The following approach computes T[i], an array that stores the maximum profit achieved from rope of length i where $1 <= i <= n$, and uses smaller values of i already computed.

We now describe the algorithm detail:

```
def ropeCut(prices, length):
    """
    Consumes: prices -> list of prices for each length of rope
              length -> length of the rope we have
    Produces: integer
    Purpose: Determines the maximum profit possible from our given rope
    """
    # initialize array of length + 1 size
    int[] T = new int[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.

2 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)
```

3 Recurrence and Induction

1. (a) Recurrence relation for T.

   - $T(1) = c_0$
   - $T(n) = T(\frac{n}{2}) + c_1 n$

   (b) Use plug-n-chug with $n = 1, 2, 4$, and 8 to conjecture a big-O solution to the recurrence.

   (i) $T(1) = c_0$

   (ii) $T(2) = T(1) + c_1 * 2 = 2c_1 + c_0$

   (iii) $T(4) = T(2) + c_1 * 4 = 4c_1 + 2c_1 + c_0 = 6c_1 + c_0$

   (iv) $T(8) = T(4) + c_1 * 8 = 8c_1 + 6c_1 + c_0 = 14c_1 + c_0$

   We can see that T(n) is a linear function, with recurrence relation T(n) = $c_0 + (2n - 2)c_1$ which makes `transform` O(n).

2. (a) Here's a recurrence relation for a different function, S:

   - $S(1) = 1$
   - $S(n) = n^2 + S(\frac{n}{2})$

   The solution to this recurrence (for n a power of 2) is:

   $$S(n) = \frac{4n^2 - 1}{3}$$

   Proof by Induction
   Base case:

   $$S(1) = \frac{4 * 1^2 - 1}{3}$$

   $$S(1) = 1$$

   Assume true for $n = k$ (Inductive assumption):

   $$S(k) = \frac{4k^2 - 1}{3}$$

   Show true for n = 2k (General case):

   $$S(2k) = (2k)^2 + S(k) = 4k^2 + \frac{4k^2 - 1}{3} = \frac{16k^2 - 1}{3} = \frac{4(2k)^2 - 1}{3}$$

   $$S(2k) = \frac{4(2k)^2 - 1}{3}$$

   Since $P(k) \rightarrow P(2k)$, and $P(1)$ is true, $P(k)$ is true where $n = 2^k$ for some non-negative integer k.

   (b) The function S is $O(n^3)$ and $\Omega(n^2/2)$.

3. (a) A (0,2) binary tree T is one in which every node has out-degree zero or two, i.e., it has either two children or it's a leaf. Give a recursive definition of a (0,2) binary tree.

   Base case: 1 node without any children is a (0,2) tree

   Recursive definition: A (0,2) tree is a binary tree where both the left and right children are (0,2) binary trees

(b) Prove by induction that the number of nodes in a regular binary tree is one more than the number of edges.

P(n) : For a tree of size n, there are n-1 edges.
Base case: P(1) = 0. This is trivially true, because a tree with only one node has no edges

Assumption: P(k) is true, i.e for a tree of size k, there are k-1 edges

Inductive step: Prove that P(k+1) is also true:

Take any tree with $k + 1$ nodes. In any binary tree, there must be at least one leaf node. We can choose any of those leaf nodes and delete it. This means we will also delete the edge connecting it to its parent. Now we are left with a tree of size k. By our inductive assumption, this tree has $k - 1$ edges. Since we had to remove one edge (and one node) to reduce our tree of size $k + 1$ to one of size k, we have shown that the number of edges in a tree of size $k + 1$ is $k - 1 + 1 = k$.
Alternatively, consider a tree with k nodes, to add a node to the tree, it must be connected to the tree by one new edge. Now the tree with k nodes had $k - 1$ edges by the inductive assumption, therefore the addition of one node with one edge takes the number of edges to k, and the number of nodes to $k + 1$

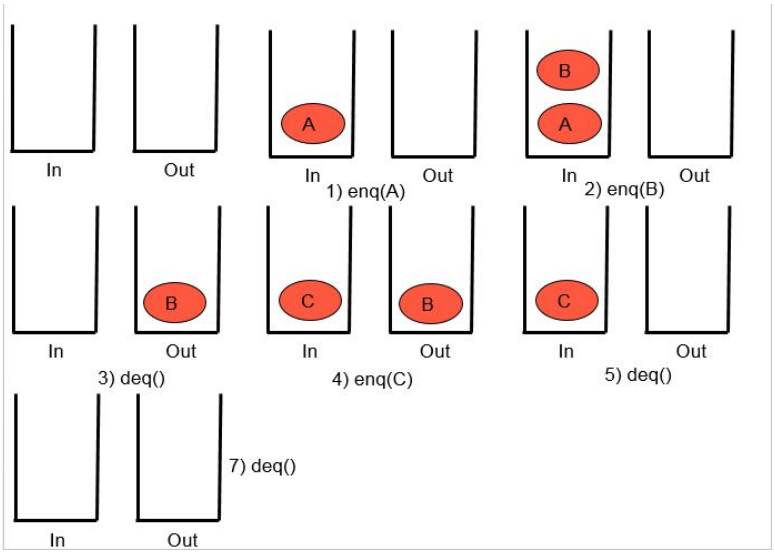
Conclusion: Since we have proven P(1) and shown that P(k)→P(k+1), we have shown that for every binary tree with one or more nodes, the number of edges is one less than the number of nodes, i.e. that P(n) is true.

4 Amortized Analysis

As you saw in Homework 1, you can implement a queue with very little additional work by using two stacks, `in` and `out`. They both start out empty. To enqueue an item, you push it onto `in`. To dequeue an item, you pop it from `out`. Of course, that depends on `out` containing something! If `out` is empty, you first "pour" all the items from `in` into `out`, and then pop from `out`. This is what "pouring" looks like:

```
def pour():
    while not in.empty():
        out.push(in.pop())
```

(a) Draw a picture to indicate the state of the stacks `in` and `out` in an empty queue to which the following operations are applied: enq(A), enq(B), deq(), enq(C), deq(), deq(). You should draw a total of seven pictures, the first and last showing two empty stacks.



(b) Explain why enqueuing is worst-case $O(1)$ and dequeuing is worst-case $O(n)$, where n is the number of items in the queue.

Enqueuing is worst case $O(1)$ because it is only a push() on the first stack (and, of course, a push is a constant time operation). Dequeuing is worst case $O(n)$ because if the out stack is empty, every element in the queue will have to be moved from the in to the out stack before it can be popped from the out stack.

(c) Explain why the amortized cost of dequeuing, in any sequence of n operations on an empty queue, is $O(1)$.

Though pour step is $O(n)$, the next n dequeue operations will be constant since we've moved those elements to the out stack. So for n operations, the amortized cost of dequeueing is $O(1)$.

5 Hashing

What is a good hash function? What's a bad one? What are hashtables and how do they work? Explain the difference between a hashset and a hashtable.

A good hash function uniformly distributes the inputs across all buckets, regardless of whether or not the input is random. A bad hash function is one that maps many inputs to afew buckets. Hashtables are arrays of arrays (or linked lists) that map keys to (key,value) pairs using a hash function. A hashset is the same as hashtable except that the value is the key - the hash function is used to determine the location where the key itself is stored.

6 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
```