Assignment 5 (SOLUTIONS)

Q1. [10 Points] Recurrence Relations *(maximum 3 pages)*

The following are some questions related to finding the closed form for various recurrence relations.

a. Use the recursion tree method to guess an asymptotically tight bound for the recurrence

$$T(n) = T(\alpha n) + T((1-\alpha)n) + cn,$$

where α is a constant in the range $0 < \alpha < 1$ and $c > 0$ is also a constant and *formally prove it is correct* using the substitution method.

Solution. The closed form expression for $T(n)$ is $T(n) = \Theta(n \log n)$. Wlog, assume that $\alpha \in (0, 1/2]$ (otherwise exchange α with $1-\alpha$). Guess that $T(n) = \Theta(n \log n)$ based on the following recurrence tree shown in Figure 1. Note that the work done on each level is cn and the height of the tree, h, satisfies $c_1 \log_\alpha n \leq h \leq c_2 \log_{(1-\alpha)} n$ for constants c_1 and c_2.

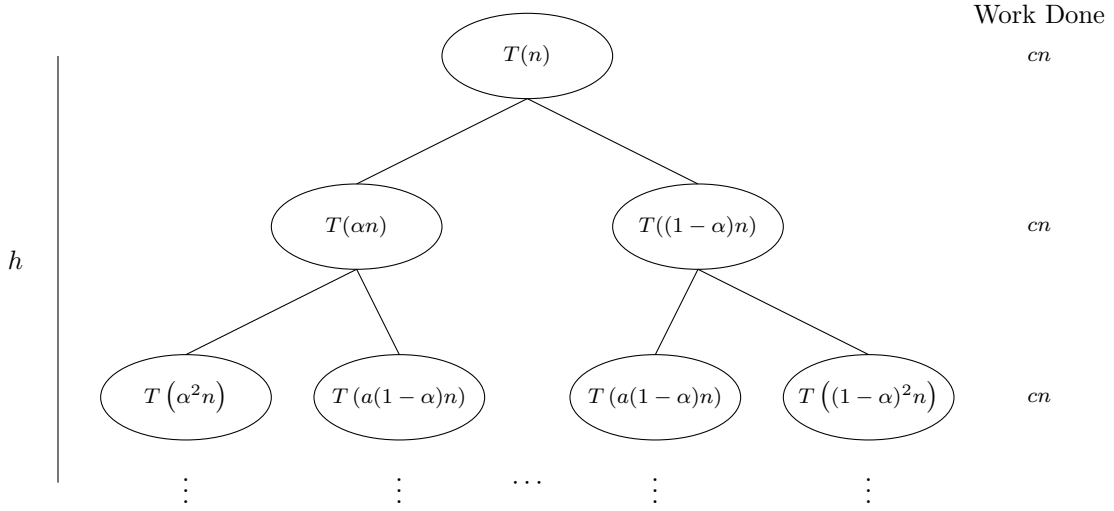

Figure 1: Recurrence tree for $T(n)$ with height h satisfing $c_1 \log_{1-\alpha} n \leq h \leq c_2 \log_\alpha n$.

To formally prove $T(n) = \Theta(n \log n)$, show: (1) $T(n) = \Omega(n \log n)$ and (2) $T(n) = O(n \log n)$ with induction. The bases of the logarithm will $1/(1-\alpha)$.

$T(n) = \Omega(n \log n)$ $P(n) :=$ there exists a constant c_1 such that $T(n) \geq c_1 n \log n$. We will constrain c_1 at the end. In the base case $T(n)$ for a constant n is always a constant so we can assume that c_1 is no larger than $T(0), ..., T(\lceil 1/\alpha \rceil)$.

In the inductive step fix $k > \lceil 1/\alpha \rceil$ and suppose that $P(0) \cdots P(k-1)$ is true. Show that $P(k)$ is true. From the definition of $T(k)$, we have $T(k) = T(\alpha k) + T((1-\alpha)k) + ck$.

By IH, we have $T(\alpha k) \geq c_1 \log(\alpha k)$ and $T((1-\alpha)k) \geq c_1 \log(1-\alpha)k$. Together,

$$\begin{aligned} T(k) = T(\alpha k) + T((1-\alpha)k) + ck &\geq c_1 \alpha k \log(\alpha k) + c_1(1-\alpha)k \log((1-\alpha)k) + ck \\ &\geq c_1 k \log\left(\alpha^\alpha (1-\alpha)^{1-\alpha} k\right) + ck \\ &\geq c_1 k \log \left(\frac{k}{\left(\frac{1}{\alpha}\right)^\alpha \left(\frac{1}{1-\alpha}\right)^{1-\alpha}} \right) + ck \\ &= c_1 k \log k - c_1 k \log \left(\left(\frac{1}{\alpha}\right)^\alpha \left(\frac{1}{1-\alpha}\right)^{1-\alpha} \right) + ck \\ &\geq c_1 k \log k \end{aligned}$$

where the last inequality is true when c_1 such that $c_1 \leq \frac{c}{\log\left(\left(\frac{1}{\alpha}\right)^\alpha \left(\frac{1}{1-\alpha}\right)^{1-\alpha}\right)}$.

$T(n) = O(n \log n)$ The process for proving the upper bound is quite similar. Consider predicate $Q(n) :=$ there exists a constant c_2 such that $T(n) \leq c_2 n \log n$. The base case is similar to before as T applied to a constant is constant. In the inductive step, we fix a $k > \lceil 1/(1-\alpha) \rceil$ and apply IH on $T(\alpha k)$ and $T((1-\alpha)k)$ to obtain

$$\begin{aligned} T(k) &\leq c_2 \alpha k \log(\alpha k) + c_2(1-\alpha)k \log((1-\alpha)k) + ck \\ &\leq c_2 k \log\left(\alpha^\alpha (1-\alpha)^{1-\alpha} k\right) + ck \\ &\leq c_2 k \log \left(\frac{k}{\left(\frac{1}{\alpha}\right)^\alpha \left(\frac{1}{1-\alpha}\right)^{1-\alpha}} \right) + ck \\ &= c_2 k \log k - c_2 k \log \left(\left(\frac{1}{\alpha}\right)^\alpha \left(\frac{1}{1-\alpha}\right)^{1-\alpha} \right) + ck \\ &\leq c_2 k \log k \end{aligned}$$

where, the last inequality is true if we pick c_2 carefully, namely $c_2 \geq \frac{c}{\log\left(\left(\frac{1}{\alpha}\right)^\alpha \left(\frac{1}{1-\alpha}\right)^{1-\alpha}\right)}$.

[5 Marks] One point for correct closed form and two points for each bound (upper and lower).

b. Some recurrence relation cannot be solved using the Master Method. For example, for

$$T(n) = 2T(n/2) + n \log_2 n$$

none of the three cases of the Master Method were appropriate.

Instead, use the recurrence tree method and give the asymptotically tight bound for the above recurrence relation. *You only need to formally prove that the lower bound is tight it formally.*

Solution. Apply the solution of the next part to obtain $T(n) = \Theta\left(n \log^2 n\right)$.

[1 Marks] One point for the correct closed form.

c. Do the same as the previous part for the following recurrence relation

$$T(n) = 2T(n/2) + n \log^k n \tag{1}$$

where $k \in \mathbb{N}$ is a fixed constant with $k \geq 1$.

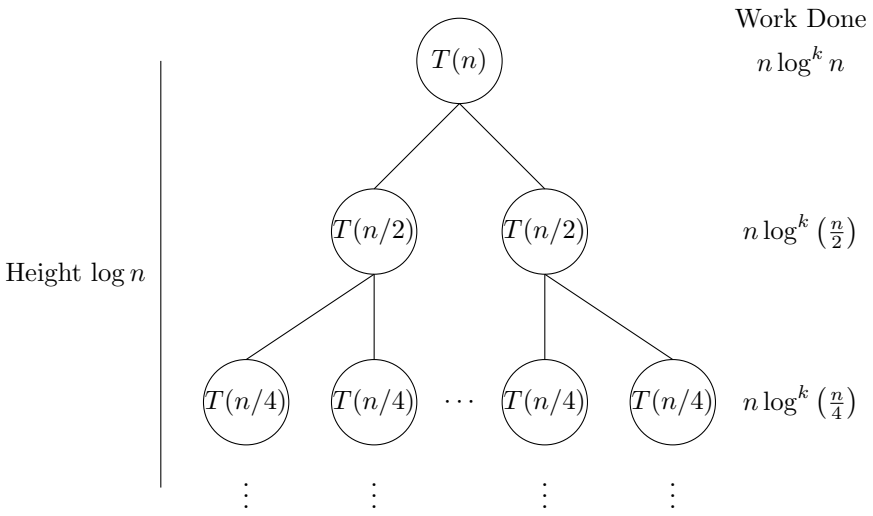

Figure 2: Recurrence tree for $T(n)$ in Equation 1.

Solution. The closed form expression for recurrence relations $T(n)$ of the form shown in Equation 1 is $\Theta\left(n \log^{k+1} n\right)$. Our guess is inspired by the recurrence tree show in Figure 2. We will only prove the lower bound is true formally. All logs in the following are base two.

Let $P(n)$ be the predicate $T(n) \geq c_1 n \log^{k+1} n$ *for constant* c_1. As before, we can assume that $T(n)$ is a small constant for all small constants n. In the inductive step, we fix a $u \geq 0$ and assume that $T(0) \wedge \cdots \wedge T(u-1)$ are true. By definition $T(u) = 2T(u/2) + u \log^k u$. By IH, we have that $T(u/2) \geq \frac{c_1 u}{2} \log^{k+1} (u/2) + \frac{c_2 u}{2}$. Putting this together, we have

$$\begin{aligned}
T(u) &= 2T(u/2) + u \log^k u \\
&\geq c_1 u \log^{k+1} \left(\frac{u}{2}\right) + u \log^k u \\
&= c_1 u \left(\log u - 1\right)^{k+1} + u \log^k u \\
&= c_1 u \left(\log^{k+1} u - \binom{k+1}{1} \log^k u + \binom{k+1}{2} \log^{k-1} u - \cdots + (-1)^{k+1}\right) \\
&= c_1 u \log^{k+1} u - \left((k+1)c_1 - 1\right) u \log^k u + c_1 u \left(\binom{k+1}{2} \log^{k-1} u - \cdots + (-1)^{k+1}\right) \\
&\geq c_1 u \log^{k+1} u
\end{aligned}$$

where the last inequality follows when $c_1 \leq \frac{1}{k+1}$.

[4 Marks] One point for the correct closed form and four points for the lower bound.

Q2. [10 Points] Dynamic Programming *(maximum 2 pages)*

This is a preview of a topic related to recursive algorithms that you will learn more about in CSC373. In this question we consider the *rod-cutting* problem.

You operate a company which buys n inches long steel rods, cuts them into shorter rods (or leaves them the same length), and then sell the cut segments. All cuts are free. Your job is to determine the best way to cut the rods. You are given as input a pricing table which lists the price p_i you can sell an i inch long rod (rod lengths are always an integral number of inches).

Table 1 gives an example pricing table for $n = 4$. Note that there are many different ways to cut the rod. If it is sold as is (no cuts) then we get \$8. If we cut it into two pieces of length one and three respectively, we would get \$7. If we cut it into two pieces both two inches long, we would get \$10. If we cut it into four pieces each of which is one inch long then we would get \$4. It follows, for this pricing table, that the optimum value \$10 is obtained by cutting the full 4 inch rod into two pieces both two inches in length.

Length i	1	2	3	4
Price p_i	1	5	6	8

Table 1: Example pricing table for input rods of length 4.

a. Consider the following *greedy algorithm*[1]. We compute the value per unit length (e.g., for the pricing table shown in Table 1, these "per-unit values" are shown in Table 2) and cut off a length with maximum price per unit length which fits in the remaining part of the rod.

Length i	1	2	3	4
Price p_i per unit length	1	2.5	2	2

Table 2: Price per unit length for input rods of length 4 and pricing table as shown in Table 1.

For the pricing table shown in Table 1, the greedy algorithm would return the optimum cut into two rods of length two inches.

Give a counter-example for the optimality of this algorithm, i.e., give a pricing table where the greedy algorithm *does not* obtain the best value.

Solution. It suffices to make a slight change to Table 1 to obtain a counter example. Consider Table 3. The optimum value is still \$10 which is achieved by cutting a 4 inch rod into two 2 inch pieces. However the greedy algorithm would cut off a 3 inch rod and then leave the remaining 1 inch rod to obtain a value of \$9.5.

Length i	1	2	3	4
Price p_i	0.5	5	9	6

Table 3: Counter example for the greedy algorithm.

[2 Marks] One for a valid counter-example and one for justifications.

b. For this part and the next part, define $OPT(i)$ to be the maximum value we can obtain by cutting a rod which is i inches long. Let n be the length of the uncut rod in inches and $p_1, ..., p_n$ be the value of selling rods of different lengths (i.e., $p_i > 0$ is the value of selling a rod of length i). What is $OPT(1)$ with respect to the p_is?

Solution. $OPT(1) = p_1$ is the profit of selling a unit length rod (prices are non-negative).

[1 Mark]

[1] *Greedy algorithms* are a class of algorithms where we assign a value to each element of the input and at each iteration take the element which has the best value subject to some constraint. Prims' algorithm, that we saw in-class, is a greedy algorithm as we always add the *lightest weight edge* subject to the constraint that it is incident to one of the vertices currently in our tree.

c. Express $OPT(k+1)$ with respect to $OPT(1), ..., OPT(k)$ and the p_is. Justify your recurrence relation.

Solution.

$$OPT(k+1) = \begin{cases} p_1 & \text{if } k = 0 \\ \max(p_{k+1}, \max_{1 \le i \le k}(p_i + OPT(k+1-i))) & \text{otherwise} \end{cases}$$

In the base case, as stated before, if $k = 0$, then the optimum value is obtained by returning the unit length rod as is (all rod length requirements are of integer length). Otherwise, we could return the length $k+1$ rod as is, obtaining a profit of p_{k+1}, or cut off any amount i, sell the rod of length i to obtain p_i profit, and obtain the maximum profit of the remaining rod of length $k+1-i$. Note that this value is *exactly* captured by $OPT(k+1-i)$.

[2 Marks] One mark for the recurrence relation and one mark for the justification. The marks for the recurrence relations should be for the non-base case.

d. *Memoization*[2] is a technique of storing pre-computed values so that the same computation executed later runs in constant time. This is typically implemented by keeping a global look-up table which all recursive calls can update. Use memoization and the previous two parts to come up with an algorithm which solves the rod-cutting problem.

Write your pseudo-code in Algorithm 1 (you can change the input and output parameters to suit your needs).

Algorithm 1 CutRod(P : price list)

Require: P is an one-indexed list of length n. For all $i \in [n]$, $P[i]$ represents the profit from selling a rod of length i.

Ensure: Return the set of cuts which results in the maximum profit.

```
T ← {}                                  ▷ look-up table storing for each i the best cut
OPT ← {}                                ▷ look-up table storing for each i the profit of the maximum cut
T[1] ← {1}
OPT[1] ← P[1]
CutRodHelper(n, P, T, OPT)
return T[n]
```

[3 Marks] One mark base case, one mark recursive call, and one mark to make sure the algorithm returns the correct answer.

e. Determine the upper bound on the running time of your algorithm by determining (1) the size of the look-up table T and (2) the time it takes to fill an entry of T.

Solution. Most of the work is done in Algorithm 2 with only constant work done in Algorithm 1. In the helper-function, we compute each entry of T and OPT by looking up values already stored in T and OPT or in P. Note that at the end of the procedure, there are n entries in both T and OPT and an upper bound on the time it takes to compute an entry is $O(n)$. It follows that the upper bound on the running time is $O(n^2)$.

[2 Marks] One mark for the correct number of entries in T and OPT and the other for the correct amount of time to compute each entry. If solution is $\omega(n^2)$ then they would have computed one of the above incorrectly and the appropriate marks should be deducted.

[2]Read more about memoization. You can efficiently memoize any function in Python using a decorator.

Algorithm 2 CutRodHelper(i : int, P : price list, T : look-up table, OPT : look-up table)

Require: i is a positive integer, with $i \leq |P|$.
Ensure: Fills out T and OPT according to specifications.

```
1: if i = 1 then return
2: end if
3: m ← P[i]
4: C ← [i]
5: for j = 1..i − 1 do
6:     if j ∉ OPT then
7:         CutRodHelper(i − j, P, T, OPT)
8:     end if
9:     if m < OPT[i − j] + P[j] then
10:        m ← OPT[i − j] + P[j]
11:        C ← T[i − j].append(j)          ▷ pointer to T[i − j] then appends to that list
12:    end if
13: end for
14: OPT[i] ← m
15: T[i] ← C
```